C Library Guide and Compiler Reference

IBM Personal Computer XENIX™ Software Development System Version 2.00

Programming Family

NOT
FOR SALE
For Demonstration
or Educational
Purposes Only
D 284556

IBM

**Personal
Computer
Software**

XENIX is a trademark of
Microsoft Corporation.

59X8657

C Library Guide and Compiler Reference

IBM Personal Computer XENIX™ Software Development System Version 2.00

Programming Family

Personal Computer Software

XENIX is a trademark of
Microsoft Corporation.

Second Edition (April 1986)

Changes are made periodically to the information herein; these changes will be incorporated in new editions of this publication. This edition applies to Version 2.00 of the IBM Personal Computer XENIX Software Development System, and to all subsequent releases until otherwise indicated in new editions or technical newsletters.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

International Business Machines Corporation provides this manual "as is," without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

Products are not stocked at the address given below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM Personal Computer dealer.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to IBM Corporation, Department 997, 11400 Burnet Road, Austin, Texas 78758. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Portions of the code and documentation described in this book were developed at the Electrical Engineering and Computer Science Department at the Berkeley campus of the University of California under the auspices of the Regents of the University of California.

© Copyright International Business Machines Corporation 1984, 1986
© Portions copyright by Microsoft Corporation 1983, 1984, 1986, all rights reserved
© Portions copyright by AT&T Bell Laboratories 1983, 1984, 1986, all rights reserved

Contents

Chapter 1. cc: A C Compiler **1-1**

Invoking the C Compiler 1-1

Creating Programs from C Source Files 1-2
  Compiling a C Source File 1-2
  Compiling Several Source Files 1-3
  Naming the Output File 1-5

Creating Small, Middle, Large, and Huge Programs . 1-5
  Small Model Programs: the -Ms Option 1-6
  Pure-Text Small Model Programs: the -Ms and -i Options 1-6
  Middle Model Programs: the -Mm Option 1-7
  Large Model Programs: the -Ml Option 1-7
  Huge Model Programs: the -Mh Option 1-8

Using Object Files and Libraries 1-9
  Creating Object Files 1-10
  Creating Programs from Object Files 1-10
  Linking a Program to Functions in Libraries 1-11
  Controlling the Version Number 1-12
  Generating a Source File Listing 1-12

Creating Smaller and Faster Programs 1-12
  Creating Optimized Object Files 1-12
  Stripping the Symbol Table 1-13
  Removing Stack Probes from a Program 1-14

Preparing Programs for Debugging 1-14
  Producing an Assembler Language Listing 1-14
  Profiling a Program 1-15

Controlling the C Preprocessor 1-16
  Defining a Macro 1-16
  Defining Include Directories 1-17
  Ignoring the Default Include Directories 1-17
  Saving a Preprocessed Source File 1-18

Error Messages 1-18
  C Compiler Messages 1-19
  Setting the Level of Warnings 1-19

Using Advanced Options 1-20
  Creating Programs from Assembler Language Source Files 1-20
  Using the near, far, and huge Keywords 1-21
  A Common Large and Huge Model Programming Error . 1-22

  Setting the Stack Size 1-23
  Using Modules, Segments, and Groups 1-24
  Creating Programs for DOS 1-25
Compiler Summary 1-26
  The cc Options 1-26
  Memory Models 1-29
  Pointer and Integer Sizes 1-29
  Segment and Module Names 1-30

Chapter 2. Include Files 2-1
/usr/include Files 2-1
/usr/include/sys Files 2-6

Chapter 3. Using the Standard I/O Functions 3-1
Preparing for the I/O Functions 3-1
  Special Names 3-1
  Special Macros 3-2
Using Command Line Arguments 3-2
Using the Standard Files 3-4
  Reading from the Standard Input 3-4
  Writing to the Standard Output 3-7
  Redirecting the Standard Input 3-9
  Redirecting the Standard Output 3-10
  Piping the Standard Input and Output 3-10
  Program Example 3-11
Using the Stream Functions 3-12
  Using File Pointers 3-12
  Opening a File 3-13
  Reading a Single Character 3-14
  Reading a String from a File 3-15
  Reading Records from a File 3-15
  Reading Formatted Data from a File 3-16
  Writing a Single Character 3-17
  Writing a String to a File 3-18
  Writing Formatted Output 3-19
  Writing Records to a File 3-19
  Testing for the End of a File 3-20
  Testing for File Errors 3-21
  Closing a File 3-22
  Program Example 3-22
Using More Stream Functions 3-24
  Using Buffered Input and Output 3-24
  Reopening a File 3-25
  Setting the Buffer 3-25

Putting a Character Back into a Buffer 3-26
Flushing a File Buffer 3-27
Using the Low-Level Functions 3-28
Using File Descriptors 3-29
Opening a File 3-29
Reading Bytes from a File 3-30
Writing Bytes to a File 3-31
Closing a File 3-31
Program Examples 3-32
Using Random Access I/O 3-35
Moving the Character Pointer 3-35
Moving the Character Pointer in a Stream 3-36
Rewinding a File 3-37
Getting the Current Character Position 3-37

Chapter 4. Screen Processing 4-1

Terminology 4-1
Screen Processing Overview 4-3
Using the Library 4-4
Screen Updating 4-5
Naming Conventions 4-6
Preparing the Screen 4-7
Initializing the Screen 4-7
When to Use initscr() 4-7
Using Terminal Capability and Type 4-8
Capabilities from termcap 4-9
Using Default Terminal Modes 4-10
Using Default Window Flags 4-10
Using the Default Terminal Size 4-11
Terminating Screen Processing 4-11
How to Use the Screen Package 4-12
Output 4-12
Input 4-13
Using the Standard Screen 4-13
Adding a Character 4-13
Adding a String 4-14
Printing Strings, Characters, and Numbers 4-14
Reading a Character from the Keyboard 4-15
Reading a String from the Keyboard 4-16
Reading Strings, Characters, and Numbers 4-16
Moving the Current Position 4-17
Inserting a Character 4-18
Inserting a Line 4-19
Deleting a Character 4-19
Deleting a Line 4-20
Clearing the Screen 4-20

Clearing a Part of the Screen 4-21
Refreshing from the Standard Screen 4-21
Creating and Using Windows 4-22
Creating a Window 4-22
Creating a Subwindow 4-23
Adding and Printing to a Window 4-24
Reading and Scanning for Input 4-25
Moving the Current Position in a Window 4-27
Inserting Characters and Lines 4-27
Deleting Characters and Lines 4-28
Clearing the Window 4-29
Refreshing from a Window 4-30
Overlaying Windows 4-31
Overwriting a Screen 4-31
Moving a Window 4-32
Reading a Character from a Window 4-32
Touching a Window 4-33
Deleting a Window 4-34
Using Other Window Functions 4-34
Drawing a Box 4-34
Displaying Bold Characters 4-35
Restoring Normal Characters 4-36
Getting the Current Position 4-36
Setting Window Flags 4-37
Scrolling a Window 4-38
Combining Movement with Action 4-38
Controlling the Terminal 4-39
Terminal Modes 4-39
Setting a Terminal Mode 4-39
Clearing a Terminal Mode 4-40
Moving the Terminal's Cursor 4-41
Getting the Terminal Mode 4-41
Variables Set by gettmode () 4-41
Saving and Restoring the Terminal Flags 4-42
Setting a Terminal Type 4-42
Variables Set by setterm() 4-42
Reading the Terminal Name 4-44

Chapter 5. Character and String Processing 5-1
Using the Character Functions 5-1
Testing for an ASCII Character 5-1
Converting to ASCII Characters 5-2
Testing for Alphanumerics 5-3
Testing for a Letter 5-3
Testing for Control Characters 5-3
Testing for a Decimal Digit 5-4

  - Testing for a Hexadecimal Digit 5-4
  - Testing for Printable Characters 5-5
  - Testing for Punctuation 5-5
  - Testing for Whitespace 5-5
  - Testing for Case in Letters 5-6
  - Converting the Case of a Letter 5-6
- Using the String Functions 5-7
  - Concatenating Strings 5-7
  - Comparing Strings 5-8
  - Copying a String 5-9
  - Getting the Length of a String 5-9
  - Concatenating Characters to a String 5-10
  - Comparing Characters in Strings 5-10
  - Copying Characters to a String 5-11
  - Reading Values from a String 5-12
  - Writing Values to a String 5-12

Chapter 6. Using Process Control 6-1
- Using Processes 6-1
- Calling a Program 6-2
- Stopping a Program 6-3
- Starting a New Program 6-3
- Executing a Program Through a Shell 6-6
- Duplicating a Process 6-6
- Waiting for a Process 6-7
- Inheriting Open Files 6-8
- Program Example 6-9

Chapter 7. Creating and Using Pipes 7-1
- Opening a Pipe to a New Process 7-2
- Reading and Writing to a Process 7-2
- Closing a Pipe 7-3
- Opening a Low-Level Pipe 7-3
- Reading and Writing to a Low-Level Pipe 7-4
- Closing a Low-Level Pipe 7-5
- Program Examples 7-6

Chapter 8. Using Signals 8-1
- Using the Signal Function 8-2
  - Disabling a Signal 8-2
  - Restoring a Signal's Default Action 8-3
  - Catching a Signal 8-4
  - Restoring a Signal 8-6
  - Program Example 8-6

Controlling Execution with Signals 8-7
Delaying a Signal's Action 8-8
Using Delayed Signals with System Functions 8-9
Using Signals in Interactive Programs 8-9
Using Signals in Multiple Processes 8-11
Protecting Background Processes 8-12
Protecting Parent Processes 8-13

Chapter 9. Using System Resources 9-1
Introduction 9-1
Allocating Space 9-1
Allocating Space for a Variable 9-2
Allocating Space for an Array 9-2
Reallocating Space 9-3
Freeing Unused Space 9-4
Locking Files 9-5
Preparing a File for Locking 9-5
Locking a File 9-6
Program Example 9-6
Header Files Needed for Calls 9-7
Semaphore Calls 9-8
Structure of a Semaphore Set 9-9
How to Use Semaphores 9-10
Example of Semaphores 9-11
Example of Semaphore Programming 9-15
Message Calls 9-19
Terms 9-20
General Operation 9-21
Using Message Queues 9-22
Example of Message Queue Calls 9-23
Shared Memory Calls 9-25
Terms 9-26
Using Shared Segments 9-27

Chapter 10. Error Processing 10-1
Using the Standard Error File 10-1
Using the errno Variable 10-2
Printing Error Messages 10-3
Encountering System Errors 10-4
Using Error Signals 10-5

Appendix A. IBM Personal Computer XENIX C Language A-1
Data Types A-1
  Structures A-2
  Bit Fields A-2
  Unions A-2
Constants A-2
String Literals A-2
Escape Sequences A-3
Reserved Words A-4
Identifiers A-4
Register Variables A-4
Right Shift Operator A-5
sizeof Operator A-5
Pointer Operator A-5
switch Statements A-5
Data Layout A-5
Preprocessor Control Lines A-6
Run-Time Limits A-7
Other Limitations A-7

Appendix B. A Common Library for PC XENIX and DOS B-1
Common Routines B-1
Differences between Common Routines B-3
DOS-Specific Routines B-10
Common Variables B-10
DOS-Specific Variables B-11
Common Include Files B-11
DOS-Specific Include Files B-11

Appendix C. Assembler Language Interface C-1
C Calling Sequence C-1
Entering an Assembler Routine C-2
Return Values C-2
Exiting a Routine C-3
Program Example C-3

Appendix D. C Compiler Messages D-1
Compiler Internal Error Messages D-2
Command Line Messages D-3
Warning Error Messages D-7
Fatal Error Messages D-18
Compilation Error Messages D-23
Run-Time Library Error Messages D-43

Index X-1

Ordering Additional Copies of This Book

To order additional copies of this publication (*IBM Personal Computer XENIX C Library Guide and Compiler Reference*) use either of the following sources:

- To order from your IBM representative, use Order Number SV21-8079.
- To order from your IBM dealer, use Part Number 59x9940.

This publication includes the following:

- *IBM Personal Computer XENIX C Library Guide and Compiler Reference*
- *IBM Personal Computer XENIX Macro Assembler Reference*
- Binder
- Slipcase.

Chapter 1. cc: A C Compiler

This chapter explains how to use the **cc** command. In particular, it explains how to:

- Compile C language source files
- Choose a memory model for a program
- Use object files and libraries with a program
- Create smaller and faster programs
- Prepare C programs for debugging
- Control the C preprocessor.

This chapter describes the error and warning messages generated by the C compiler. It also explains how to use advanced features of the **cc** command to make customized programs.

This chapter assumes that you are familiar with the C programming language. It also assumes that you can create C program source files using a PC XENIX text editor.

Invoking the C Compiler

The **cc** command has the form:

`cc [`*option*`]` . . . *filename* . . .

where *option* is a command option, and *filename* is the name of a C language source file, an assembler language source file, an object file, or an archive library. You can give more than one option or file name. You must separate each item with one or more spaces.

The **cc** command options let you control and modify the tasks performed by the command. For example, you can direct **cc** to perform optimization or create an assembler listing file. The options also let you specify additional information about the compilation. Many options are described in the following sections. For a complete description of all options, see **cc** in the *XENIX Commands Reference.*

Creating Programs from C Source Files

The **cc** command creates executable programs from C language source files. You can identify the contents of a file by the file name extension. For the **cc** command, C source files must have the extension **.c**.

The **cc** command can create executable programs only from source files that make up a complete C program. A complete program must have one (and only one) function named **main**. This function becomes the entry point for program execution. The **main** function can call other functions if you define them within the program or they are part of the C standard library.

Compiling a C Source File

You can compile a C source file by giving the name of the file when you invoke the **cc** command. The command compiles the statements in the file, then copies the executable program to the default output file **a.out**.

To compile a source program, enter:

`cc` *filename*

where *filename* is the name of the file containing the program. The program must be complete; that is, it must contain a **main** program function.

For example, assume that the file **main.c** stores the following program:

```
#include <stdio.h>

main()
{
        int x, y;

        scanf("%d  %d", &x, &y);
        printf("%d\n", x+y);
}
```

To compile this program, enter:

```
cc main.c
```

The command first invokes the C preprocessor, which adds the statements in the file **/usr/include/stdio.h** to the beginning of the program. It then compiles these statements and the rest of the program statements. Next, the command links the program with the standard C library. The library contains the object files for the **scanf** and **printf** functions. Finally, it places the executable code in the file **a.out**.

You can then execute the new program by entering:

```
a.out
```

The program waits until you enter two numbers, then prints their sum. For example, if you type `3 5` the program displays `8`.

Compiling Several Source Files

Large source programs are often split into several files to make them easier to understand, update, and edit. You compile such a program by giving the names of all the files belonging to the program when you invoke the **cc** command. The command reads and compiles each file in turn. Then it links all object files together and copies the new program to the file **a.out**.

To compile several source files, enter:

cc *filename filename* . . .

where you separate each *filename* from the next by at least one space. One of these files (and only one) must contain a **main** function. The others can contain functions called by this **main** function or by other functions in the program. The files must not contain calls to functions not explicitly defined by the program or by the standard C library.

For example, suppose the file **main.c** stores the following **main** program function:

```
#include <stdio.h>
extern int add();

main()
{
        int x, y, z;

        scanf("%d  %d", &x, &y);
        z = add(x, y);
        printf("%d \n", z);
}
```

Assume that the file **add.c** stores the following function:

```
add(a, b)
int a, b;
{
        return(a + b);
}
```

You can compile these files and create an executable program by entering:

```
cc main.c add.c
```

The command compiles the statements in **main.c**, then compiles the statements in **add.c**. Finally, it links the two together (along with the standard C library), and places the executable code in the file **a.out**. This program, like the program in the previous section, waits for two numbers, then prints their sum.

The **cc** command cannot keep track of more than one compiled file. When you compile several source files at a time, the command creates object files to hold the binary code generated for each source file. Then it links these object files to create an executable program. The object files have the same base name as the source files, but they have the **.o** file extension. For example, when you compile the two source files above, the compiler produces the object files **main.o** and **add.o**. These files are permanent files; that is, the command does not delete them after completing its operation. The command also creates an object file if it compiles only one source file.

Naming the Output File

You can give the executable program file any valid file name by using the **-o** (for output) option. The option has the form:

```
-o filename
```

where *filename* is a valid file name or path name.

If you give a file name, the program file is stored in the current directory. If you give a full path name, the file is stored in the given directory. If that file already exists, the new executable program replaces its contents.

For example, the command:

```
cc  -o addem  main.c  add.o
```

causes the compiler to create an executable program file **addem** from the source file **main.c** and object file **add.o**. You can execute this program by entering:

```
addem
```

The **-o** option does not affect the existing **a.out** file. This means that the **cc** command does not change the current contents of **a.out** if you give the **-o** option.

Creating Small, Middle, Large, and Huge Programs

C programs in memory consist of the actual machine instructions created from the source statements. They also consist of data storage created for the variables. The data storage also contains the stack used by the programs for temporary storage during execution. PC XENIX stores the instructions and data in one or more segments of physical memory. Each segment is 64K bytes. The maximum allowable size for any program depends on how many segments PC XENIX allocates for it when it is compiled.

The **cc** command lets you create programs of a variety of sizes and purposes using the **-Ms**, **-Mm**, **-Ml**, **-Mh**, and **-i** options. These options define the number of memory segments PC XENIX allocates for your program to use. They also determine how the system loads the program for execution.

The **cc** command allows you to create programs in the following four memory models:

- Small model (**Ms**)
- Middle model (**Mm**)
- Large model (**Ml**)
- Huge model (**Mh**).

In addition, you can define small model programs as either pure or impure. Each model defines a different type of program structure and storage.

Following are descriptions of the memory model options.

Small Model Programs: the -Ms Option

You can create a small model program by using the **-Ms** option. Impure-text small model programs are C programs that are short. These programs must not exceed 64K bytes. The **-Ms** option directs **cc** to create a program that occupies a single segment when loaded into physical memory. To create a small model program where *filename* is the name of the program you wish to compile, enter:

`cc -Ms` *filename*

The **cc** command creates small model programs by default when you do not otherwise specify a program model. Thus, the **cc** command does not require you to use the **-Ms** option.

Pure-Text Small Model Programs: the -Ms and -i Options

You can combine the **-Ms** and **-i** options to create a pure-text small model program. Pure-text small model programs are short programs that many users can invoke. Pure-text programs can occupy up to 128K bytes. You cannot, however, use more than 64K bytes each for either instructions or data. Unlike impure-text small model programs, for multiple invocations, the system loads one copy of the instructions of a pure-text program into memory. As long as this copy stays in memory, the system loads a copy of the new data for each new invocation of the program. It then keeps each copy of data separate, while sharing the instructions among the different invocations. Small model programs could waste valuable memory space without pure-text programs.

The **-i** option directs **cc** to create separate memory segments for the instructions and data of a small model program. To create a pure-text program, enter:

```
cc -Ms -i filename
```

where *filename* is the name of the file source program you want to compile. Because **cc** creates small model programs by default, you only need to use the **-i** option.

Middle Model Programs: the -Mm Option

You can create a middle model program by using the **-Mm** option. Middle model programs are C programs that have a large number of program statements but a small amount of data. Program instructions can be any size, but program data must not exceed 64K bytes. The **-Mm** option creates one segment for the data of the program and one or more segments for the instructions. To create a middle model program, enter:

```
cc -Mm filename . . .
```

where *filename* is the name of the source file you want to compile. When creating a program, the compiler attempts to fit as many instructions as possible into a segment (up to 64K bytes).

Middle model programs are pure in the sense that the system never loads more than one copy of the program's instructions into memory at one time. This means that you do not have to use the **-i** option for middle model programs.

Large Model Programs: the -Ml Option

You can create large model programs by using the **-Ml** option. Large model programs are very large C programs that use a large amount of data storage during normal processing. Program instructions and data can have any size. The program, however, must not contain a single array or structure exceeding 64K bytes. The **-Ml** option directs **cc** to create multiple segments for both instructions and data. To create a large model program, enter:

```
cc -Ml filename
```

where *filename* is the name of a source file you want to compile.

As with middle model programs, the compiler attempts to fit as many instructions into a segment as possible. Like middle model programs, large model programs are pure.

Huge Model Programs: the -Mh Option

You can create huge model programs by using the **-Mh** option. Huge model programs occupy several segments with both text and data in as many segments as necessary. With the **-Mh** option, it is possible to have an array that exceeds 64K bytes. The program, however, cannot contain a single structure exceeding 64K bytes.

We define an *item* to be the following data constructs that can be elements of an array:

- Characters
- Integers
- Floating-point numbers
- Structures
- Unions.

The huge model implementation is a compromise. The run-time cost of executing the code necessary to reference items divided across data segments would be prohibitive. The huge model implementation, therefore, has the following limitations:

- An item must not exceed a 64K byte segment (it must not cross a segment boundary). This means that the address of an item can access any part of the item.
- Each group of segments allocated to an array is 64K bytes long. An exception is the segment the system allocates to the highest part of the array. The system may not fully use this segment.
- The segments must be adjoining. On an 80286 processor, this means that the segments must have adjoining local descriptor table entries.

As a consequence of these limitations, no structure or union can be greater than 64K bytes. If an array is greater than 64K bytes but less than 128K bytes, it can be offset within a segment. This ensures that the elements align to the 64K-byte boundary. If the array requires three or more segments, the size of the elements in the array must be a power of two. This is always true for scalar elements; the compiler will flag composite elements that do not comply and you must pad them correctly.

If you declare a program huge, it affects the following address calculations:

- Pointer increment or decrement operations
- Static array indexing
- Pointers to structure elements
- Indexing into an array on a stack frame.

Each of these involves arithmetic that requires adjustment of the selector part of the address. Huge pointers and large pointers are both 32 bits long, but the system uses these bits differently in addressing memory. For the large pointer, the system uses only the low order half of the pointer (the first 16 bits). These first 16 bits reference the offset within the segment, and the system does arithmetic calculations only on them. In large pointers, the high-order 16 bits reference the segment location. In huge model pointers, though, arithmetic is done on all 32 bits. This is necessary because huge data constructs are capable of spanning segments.

To create a huge model program, enter:

`cc -Mh` *filename*

where *filename* is the name of a source file you want to compile.

Using Object Files and Libraries

The **cc** command lets you save functions as object files. You can use these object files to create programs at a later time. This saves you the time and trouble of recompiling the functions each time you need them. All object files created by **cc** have the file extension **.o**.

The **cc** command also lets you use functions found in PC XENIX system libraries, such as the standard C library or the screen processing library **curses**. To use these functions, you simply supply the name of the library containing them. In some cases, such as for the standard C library, **cc** accesses the library without you naming it.

You can create your own libraries with the **ar** and **ranlib** commands. These commands, described in the *XENIX Commands Reference*, copy your object files to a library file, and prepare the file for use by the **cc** command. You can access the library the same way you access any other library in the system if you copy it to the **/lib** directory.

Creating Object Files

You can create a linkable object file for each source file by using the **-c** (for compile) option. The **-c** option does not link these files. It does not create an executable program. This option directs **cc** to compile the source file without creating a final program. The option has the form:

```
cc -c filename . . .
```

where *filename* is the name of the source file. You can give more than one file name, but separate each name from the next by a space.

To make object files for the source files **add.c** and **mult.c**, type:

```
cc -c add.c mult.c
```

This command compiles each file and copies the compiled source files to the object files **add.o** and **mult.o**. It does not link these files; it does not create an executable program.

The **-c** option saves useful functions for programs developed later. Once a function is in an object file you can use it as is, or you can save it in a library file and access it like other library functions. We describe this in the following sections.

The **cc** command creates object files for each source file in the command line. Unless you give the **-c** option, however, it also attempts to link these files, even if they do not form a complete program.

Note: For more information about the **cc** command, see **cc** in the *XENIX Commands Reference.*

Creating Programs from Object Files

You can use the **cc** command to create executable programs from one or more object files, or from a combination of object files and C source files. The command compiles the source files (if any), then links the compiled source files with the object files to create an executable program.

To create a program, give the names of the object and source files you wish to use. For example, if the source file **main.c** contains calls to the functions **add** and **mult** (saved in the object files **add.o** and **mult.o**), you can create a program by entering:

```
cc main.c add.o mult.o
```

The **cc** command compiles **main.c**, then links it with **add.o** and **mult.o** to create the executable file **a.out**.

Linking a Program to Functions in Libraries

You can link a program to functions in a library by using the **-l** (lowercase el, for library) option. The option directs **cc** to search the given library for the functions called in the source file. If **cc** finds the functions, the command links them to the program file.

The option has the form:

```
cc -lname
```

where *name* is a short version of the library's actual file name. Spaces between the name and option are optional. The linker searches the **/lib** directory for the library.

For example, the command:

```
cc main.c -lcurses
```

accesses the library **/lib/Slibcurses.a** while compiling the source file **main.c**.

A library is a convenient way to store a large collection of object files. PC XENIX provides several libraries, the most common of which is the standard C library. Functions in this library are linked to your program whenever you invoke the compiler. To link other libraries, such as **libcurses.a**, use the **-l** option.

A library search for an **extern** variable loads only a module in which the variable is explicitly initialized at the external level. That is the unique defining instance of the variable.

In general, the **cc** command does not search a library until it encounters the **-l** option. Therefore, placement of the option is important. The option must follow the names of any source files containing calls to functions in the given library. In general, place all library options at the end of the command line, after all source and object files.

Controlling the Version Number

Use the **-V** option for version control of your programs. For example, the command:

```
cc -Vstring main.c
```

copies *string* to the object file of the program **main**.

Generating a Source File Listing

The **-Fs** option instructs the compiler to generate a source file listing during compilation. It numbers the listing by lines. The listing contains symbol table definitions. For example, the command:

```
cc -Fs main.c
```

produces a source file listing of the program **main**. The listing file has the extension **.S**.

Creating Smaller and Faster Programs

A C program can sometimes be made more efficient by using the optimizing options available with the **cc** command. These options can reduce the size of a compiled program by removing unnecessary or redundant instructions or unnecessary symbol information. Smaller programs save valuable space and sometimes run faster. Using the optimizing options, however, increases the compile time.

Creating Optimized Object Files

You can create an optimized object file or an optimized program from a given source file by using the **-O** (for optimize) option. For example, the command:

```
cc -O main.c
```

creates an optimized program from the source file **main.c**. The resulting object file (or program) may be more efficient than if you had compiled the source without the option.

The **-O** option applies to source files only. PC XENIX ignores existing object files if included with this option. The option must appear before the names of the files you wish to optimize. For example, the command:

```
cc -O main.c add.c
```

optimizes **main.c** and **add.c** to create **main.o**, **add.o**, and **a.out**.

You can combine the **-O** and **-c** options to compile and optimize source files without linking the resulting object files. For example, the command:

```
cc -O -c main.c add.c
```

creates separate optimized object files from the source files **main.c** and **add.c**.

Although optimization is very useful for large programs, it takes more time than regular compilation. Use it in the last stage of program development, after you have debugged the program.

Stripping the Symbol Table

Another way to reduce the size of a program's executable file is to use the **-s** option. This option directs **cc** to remove items from the symbol table. The symbol table contains information about code relocation and program symbols. The PC XENIX debugger, **adb**, uses this information to allow symbolic references to variables and functions when debugging. For normal execution, you do not need the information in this table. Remove the table when you have completely debugged the program.

The **-s** option strips the entire table, leaving machine instructions only. For example, the command:

```
cc -s main.c add.c
```

creates an executable program that does not contain a symbol table. It also creates the object files **main.o** and **add.o** that do not contain symbol tables.

You can combine the **-s** option with the **-O** option to create an optimized and stripped program. You also can strip a program with the **strip** command. See the *XENIX Commands Reference* for details.

Removing Stack Probes from a Program

You can reduce the size of a program slightly by using the **-K** option to remove all stack probes. A stack probe is a short routine called by a function to check the program stack for available space. You should not need probes if the program makes very few function calls or has unlimited stack space.

To remove the stack probes from the program **main.c**, enter:

```
cc -K main.c
```

Although this option, when combined with the **-O** option, makes the smallest possible program, use it with care. Removing stack probes from a program whose stack use you do not know can cause execution errors.

Preparing Programs for Debugging

The **cc** command provides a variety of options to prepare a program for debugging. These options range from creating an assembler language listing of the program for use with the debugger, **adb**, to adding routines for profiling the execution of a program.

Producing an Assembler Language Listing

You can use assembler language listing files to debug programs with **adb**, because **adb** recognizes machine instructions and object code statements.

You can generate an assembler language listing of your compiled source file by using the **-S** and **-L** options. The **-S** option creates an assembler source listing of the compiled C source file. Then it copies this listing to the file whose base name is the same as the source, but whose extension is **.s**. The **-L** option creates a listing that shows assembled code, as well as instructions. The file that **-L** creates has the file extension **.L**.

To create an assembler language listing, give the name of the desired source file. For example, the command:

```
cc -S add.c
```

creates an assembler language listing file named **add.s** and the command:

```
cc -L mult.c
```

creates a listing file named **mult.L**. Both the **-S** and **-L** options suppress subsequent compilation of the source file; they imply the **-c** option. Thus, no program file is created and no linking is performed.

The **-S** and **-L** options apply to source files only; the compiler cannot create an assembler language listing file from an existing object file. The option in the command line must appear before the names of files for which you want the assembly listing.

Profiling a Program

You can examine the flow of execution of a program by adding profiling code to the program with the **-p** option. The profiling code keeps a record of the number of times program functions are called during execution of the program. This record is written to the **mon.out** file. You can examine this file with the **prof** command.

For example, the command:

```
cc -p main.c
```

adds profiling code to the program created from the source file **main.c**. The profiling code calls the **monitor** function, which creates the **mon.out** file at normal termination of the program.

You must give the **-p** option in any command line that references object files that contain profiling code. For example, if you used the command:

```
cc -c -p f1.c f2.c
```

to create the object files **f1.o** and **f2.o**, then you must use the command:

```
cc -p f1.o f2.o
```

to create an executable program from these files.

Controlling the C Preprocessor

The **cc** command provides a number of options that let you control the operation of the C preprocessor. These options let you define macros, create new search paths for include files, and suppress subsequent compilation of the source file.

Defining a Macro

You may define the value or meaning of a macro used in a source file by using the **-D** (for define) option. This option lets you assign a value to a macro when you invoke the compiler. It is useful if you have used **#if**, **#ifdef**, and **#ifndef** directives in your source files.

The option has the form:

`-D`*name*`[=`*string*`]`

where *name* is the name of the macro and *string* is its value or meaning. If you do not give *string*, the compiler assumes the macro to be defined and sets the value to 1. For example, the command:

```
cc -DNEED=2 main.c
```

sets the macro **NEED** to the value 2. This is the same as having the directive:

```
#define NEED 2
```

as the first statement in the source file. The command compiles the source file **main.c**, replacing every **NEED** token with **2**.

The **-D** option is useful with the **#ifdef** directive. You may use the option to determine which statements in the source file **cc** will compile. For example, suppose a source file, **main.c**, contains the directive:

```
#ifdef NEED
```

but does not contain an explicit **#define** directive for the macro NEED. Then **cc** compiles all statements following the **#ifdef** directive only if you define NEED using the **-D** option. For example, the command:

```
cc -DNEED main.c
```

is sufficient to compile all statements following the **#ifdef** directive, while the command:

```
cc main.c
```

causes **cc** to ignore those statements.

You may use **-D** to define up to 10 macros on a command line. However, once you define a macro on the command line, you cannot redefine that macro on the command line. For a file to use a macro, you must place the **-D** option before that file's name on the command line. For example, the command:

```
cc main.c -DNEED add.c
```

defines the macro NEED for **add.c** but not for **main.c**.

Defining Include Directories

You may explicitly define the directories containing include files by using the **-I** (for include) option. This option adds the given directory to a list of directories where **cc** searches for include files. The compiler searches the directories in the list whenever it encounters an include directive in the source file. The option has the form:

`-I`*directoryname*

where *directoryname* is a valid path name to a directory containing include files. For example, the command:

```
cc -I/usr/joe/include main.c
```

causes the compiler to search the directory **/usr/joe/include** for include files requested by the source file **main.c**.

You can specify several directories for the compiler to search, but each directory name must be preceded by **-I**. The compiler searches the directories in the order you list them and only until it finds the given include file. The **/usr/include** directory is the default include directory. The compiler always searches this directory after directories given with **-I**.

Ignoring the Default Include Directories

You may prevent the C preprocessor from searching the default include directories by using the **-X** option. Use this option with the **-I** option to define the location of include files that have the same names as those found in the default directories, but which contain different definitions. For example, the command:

```
cc -X -I/usr/joe/include main.c add.c
```

causes **cc** to look for all include files only in the directory **/usr/joe/include**.

Saving a Preprocessed Source File

You may save a copy of the preprocessed source file by using the **-P**, **-E**, and **-EP** options. The file is identical to the original source file except that **cc** has expanded or replaced all C preprocessor directives. The **-P** option copies the result to the file named *filename*.**i**, where *filename* is the same name as the source file without the **.c** extension. The **-E** option copies the result to the standard output, and places **#line** directives in the output. You may save this output by redirecting it. For example, the command:

```
cc -P main.c
```

creates a preprocessed file **main.i** from the source file **main.c**, and the command:

```
cc -E add.c >add.i
```

creates a preprocessed file from the source file **add.c**. This command redirects the output to the file **add.i**.

The **-EP** option preprocesses each source file as described for **-E**, but it does not place **#line** directives in the output. The **-P**, **-E**, and **-EP** options suppress compilation of the source file. Thus, **cc** does not create an object file or program.

Error Messages

The C compiler generates a broad range of error and warning messages to help you locate errors and potential problems in programs. In addition to compiler messages, the **cc** command also displays error messages generated by the C preprocessor, the Macro Assembler, and the linker programs. The following sections describe the form and meaning of the compiler error and warning messages. (See Appendix D, "C Compiler Messages" for a list of the compiler messages. See *XENIX Macro Assembler Reference* for a list of linker error messages.)

C Compiler Messages

The compiler displays messages about syntactic and semantic errors in a source file, such as misplaced punctuation, illegal use of operators, and undeclared variables. It also displays warning messages about statements containing potential problems caused by data conversions or the mismatch of types. Error and warning messages have the form:

msgtype errornumber: *message*

where *msgtype* is the type of message, *errornumber* is the number of the error or warning, and *message* is a description of the error or warning.

If an error is severe, the compiler displays a message and terminates the compilation. Otherwise, the compiler continues looking for other errors, but does not create an object file. If the compiler displays only warning messages, it completes compilation and creates an object file.

You can avoid many C compiler errors by using the C program checker **lint** before compiling your C source files. The **lint** program performs detailed error checking on a source file. It provides a list of errors and possible problems that can affect execution of the program.

Setting the Level of Warnings

You may set the level of warning messages produced by the compiler by using the **-W** option. This option directs the compiler to display messages about statements that **cc** may not compile as the programmer intends. Warnings indicate potential problems rather than errors. The option has the form:

`-W` *number*

where *number* is a number in the range 0 to 3 giving the level of warnings. The default is level 1. The levels are:

Level	**Warning**
0	Suppresses all warning messages. The compiler displays only messages relating to syntactic or semantic errors.
1	Warns about potentially missing statements, nonreachable statements, and other structural problems. Also warns about overt type mismatches.

2 Warns about all type mismatches.

3 Warns on all automatic data conversions.

The higher option levels are useful in the earlier stages of program development when messages about potential problems are most helpful. The lower levels are best for compiling programs where you intentionally designed questionable statements. For example, the command:

```
cc -W 3 main.c
```

directs the compiler to perform the highest level of checking, and produces the greatest number of warning messages. The command:

```
cc -W 0 main.c
```

produces no warning messages. The **-w** option has the same effect as **-W 0**.

Using Advanced Options

The **cc** command provides a number of advanced programming options that give greater control over the compilation process. Some of these options also control final form of the executable program. The following sections describe a number of these options.

Creating Programs from Assembler Language Source Files

You may use the **cc** command to create executable programs from a combination of C source files and 8086/80286 assembler language source files. Assembler language source files must contain 8086/80286 instructions and must have the extension **.s**.

When you give assembler language source files, the **cc** command invokes the assembler, **as**, to assemble the instructions and create an object file. The object file may then be linked with object files created by the compiler. For example, the command:

```
cc  main.c  add.s
```

compiles the C source file **main.c**, but assembles the assembler language source file **add.s**. The resulting object files, **main.o** and **add.o**, are linked to form a single program.

When using assembler language routines with C programs, you must provide the correct interface for calls to and from C language functions. C functions require a specific calling and return sequence. Assembler language functions that fail to provide this interface cause errors. See Appendix C, "Assembler Language Interface" for more information.

Using the near, far, and huge Keywords

The **near**, **far**, and **huge** keywords are special type modifiers that define the length and meaning of the address of a given variable. The **near** keyword defines an object with a 16-bit address. The **far** and **huge** keywords define an object with a full 32-bit segmented address. You can access any data item, construct, or function.

The keywords override the normal address length generated by the compiler for variables and functions. Therefore, you need to enable these keywords at compile time (with the **-Me** option). In small model programs, **far** lets you access data and functions in segments outside of the program. In middle, large, and huge model programs, **near** lets you access data with just an offset. In all programs, the **huge** keyword lets you access a construct (which spans data segments) that is outside the segments containing the program.

When you use the **-Me** option, the compiler treats the words **fortran** and **pascal** as reserved words. If you use the **-Me** option, and you also use the words **fortran** and **pascal** as identifiers, you receive a syntax error.

The examples in the following table illustrate the **far** and **near** keywords as used in declarations in a small model program.

Declaration	**Address Size**	**Item Size**
char c;	near (16 bits)	8 bits (data)
char far d;	far (32 bits)	8 bits (data)
char *p;	near (16 bits)	16 bits (near pointer)
char far *q;	near (16 bits)	32 bits (far pointer)

Figure 1-1 (Part 1 of 2). Uses of near and far Keywords

Declaration	Address Size	Item Size
char * far r;	far (32 bits)	16 bits (near pointer) [1]
char far * far s;	far (32 bits)	32 bits (far pointer) [2]
int f();	near (16 bits)	function returning 16 bits
int far f();	far (32 bits)	function returning 16 bits[3]

Figure 1-1 (Part 2 of 2). Uses of near and far Keywords

Notes:
[1] This example has no meaning; it is shown for syntactic completeness only.
[2] This is similar to accessing data in a large model program.
[3] This example leads to trouble in most environments. The far call changes the CS register and makes run-time support unavailable.

The following example is from a middle model compilation:

```
int near f();
```

This does a near call in an otherwise far (calling) program.

Because there is no type checking between items in separate source files, use the **near**, **far**, and **huge** keywords with care.

A Common Large and Huge Model Programming Error

In the large and huge models where data pointers are 32 bits, pointer function arguments are automatically coerced to 32-bit pointers before they are passed to the function. This will happen even if the pointer is defined as a **near** pointer. For example, the following program will not work correctly:

```
main()
{
   int near *x;
   int y = 1;

   f(x, y);   /* x will be coerced into a far pointer. */
}

int f(p, a)
int near *p;
int a;
{
   printf("Does it equal one?  a = %d\n", a);
}
```

For correct results, you must declare `f` before calling `f`. This declaration must define the variable types that `f` will receive. For example:

```
int f(int near *, int);

main()
{
   int near *x;
   int y = 1;

   f(x, y);   /* x will not be coerced into a far pointer. */
}

int f(p, a)
int near *p;
int a;
{
   printf("Does it equal one?  a = %d\n", a);
}
```

Setting the Stack Size

You can set the size of the program stack by using the **-F** option. This option has the form:

`-F` *hexnum*

where *hexnum*, a hexadecimal number, is the size (in bytes) of the program stack. The program stack stores function parameters and automatic variables. If you do not use the option, the compiler sets a default stack size (4K bytes).

You can determine the stack requirements of a given program by using the **stackuse** command. **stackuse** analyzes C source files and computes the minimum stack requirement for all functions in the program. The program displays a warning if it encounters recursive functions. You must determine stack use requirements for recursive functions. For more information, see **stackuse** (CP) in the *XENIX Commands Reference.*

All programs created by **cc** have fixed-size stacks. This means you cannot increase the stack size during execution of the program. Therefore, you must give a sufficient stack size when compiling the program.

Using Modules, Segments, and Groups

Module is another name for the object file created by the C compiler. Every module has a name. The **cc** command uses this name in error messages if it encounters problems during linking. The module name is usually the same as the name of the source file (without the **.c** or **.s** extension). You can change this name using the **-NM** option. The option has the form:

-NM *name*

where *name* can be any combination of letters and digits. Only the first 40 characters of *name* are significant.

Changing the name of a module is useful if the source file you want to compile is the output of a program preprocessor and generator, such as **lex** or **yacc**.

A segment is a contiguous block of binary code produced by the C compiler. Every module has two segments: a text segment containing the program instructions and a data segment containing the program data. Each segment in every module has a name. The compiler uses this name to define the order in which the segments of the program appear in memory when loaded for execution. The compiler loads text segments having the same name as a contiguous block of code. It also loads data segments of the same name as contiguous blocks.

The C compiler creates text and data segment names. These default names depend on the memory model chosen for the program.

For example, in all small model programs, the compiler names the text segment **_TEXT** and the data segment **_DATA**. The compiler loads all segments from all modules of a small model program as a contiguous block. In middle model, large model, and huge model programs, each text segment has a different name. The default text and data segment names for middle, large, and huge model programs are in the section “Segment and Module Names” on page 1-30.

You can override the default names used by the C compiler (and override the default loading order) by using the **-NT** and **-ND** options. These options set the names of the text and data segments to a given name. The options have the form:

`-NT` *name*

and

`-ND` *name*

where *name* is any combination of letters and digits. Only the first 40 characters of *name* are significant.

These options are useful in middle and large model programs that have no specific loading order. In these programs, you can guarantee contiguous loading for two or more segments by giving them the same name. You cannot use the standard library routines within a module compiled with the **-ND** option.

The compiler loads all text and data segments into one or more physical segments of memory. It may or may not load them as contiguous blocks. All the segments in a physical segment are called a ***group***.

All programs have at least two groups: a text group and a data group. Each group has a name. The text group is named **IGROUP** and the data group is named **DGROUP**. The C compiler applies these names to the text and data segments in each module. When it links the modules, all text segments belong to the same group and all data segments belong to the same group.

Creating Programs for DOS

You can create IBM Personal Computer Disk Operating System (DOS) executable C programs from existing PC XENIX C language source files by using the **-dos** option. This option directs **cc** to compile and link C language source files for execution on DOS systems. The executable file that the **cc** command specified with the **-dos** option creates can be copied and then run on other DOS systems.

When you specify the **-dos** option, the compiler uses a special DOS linker, include files, and libraries to produce the executable file. A program that is compiled with the **-dos** option does not need the **-lm** option because the math library is automatically used.

Note: When you use the **-dos** option, the default for code generation is 8086/8088.

Compiler Summary

The following sections summarize some of the **cc** options and memory models. See *XENIX Commands Reference* for a complete list of **cc** options.

The cc Options

The following is a list of some **cc** command line options:

-c	Creates a linkable object file for each source file. It does not create an executable program.
-C	Preserves comments when preprocessing a file (only when used with **-P** or **-E**).
-dos	Creates an executable program for DOS. (Program requires an **.exe** extension to run under DOS.) When this option is specified, the DOS math libraries are automatically searched. Do not use this option with the **-lm** option. You can specify the following options only when you specify the **-dos** option: **-FP***string* Sets the floating-point option. You can specify any of the following for *string*: **a** Generates floating-point calls and selects the alternate math library. **c** Generates floating-point calls and selects the emulator (uses 8087/80287 if one is present). **c87** Generates floating-point calls and selects the 8087/80287 library (requires an 8087 or 80287 at run time).

	i	Generates in-line 8087/80287 instructions and selects the emulator (uses 8087 or 80287 if one is present).
	i87	Generates in-line 8087/80287 instructions and selects the 8087/80287 library (requires an 8087 or 80287 at run time).
	link*library*	Passes the specified *library* to the linker as the libraries field instead of using the default libraries.
	Zl	Removes the default library information from the object file.

-D*name***[**=*string***]**	Defines *name* to the preprocessor. The value is *string*, or 1 if *string* is not specified.
-E	Preprocesses each source file, copying the result to the standard output.
-EP	Preprocesses each source file as described for **-E**, without the **#line** directive.
-F *hexnum*	Sets the size of the program stack.
-Fs	Instructs the compiler to generate a source listing file.
-i	Creates separate instruction and data spaces for small model programs.
-imp	Combines instruction and data spaces for small model programs. (Impure-text)
-I*pathname*	Adds *pathname* to the list of directories where **cc** searches for **#include** files.
-K	Removes stack probes from a program.
-l *name*	Searches library *name* for unresolved function names.
-L	Creates an assembler listing file containing assembled code and assembler source instructions. Suppresses creation of an executable file.

-M*string*	Sets the program configuration. The *string* can be any combination of: **s** small model **m** middle model **l** large model **h** huge model **Note:** The **s**, **m**, **l**, and **h** are mutually exclusive. **e** enables **far**, **near**, and **huge** keywords, and reserves **fortran** and **pascal** keywords **2** enables 80286 code generation—this is the normal default **0** enables 8086/8088 code generation—the default when you use the **-dos** option **t** *num* sets data threshold for largest item in a segment (default is 32K bytes).
-nl *num*	Sets the maximum length of external symbols. Refer to **cc** in *XENIX Commands Reference.*
-ND *name*	Sets the data segment name.
-NM *name*	Sets the module name.
-NT *name*	Sets the text segment name.
-o *filename*	Makes *filename* the name of the executable file.
-O	Invokes the object code optimizer.
-p	Adds code for program profiling.
-P	Preprocesses source files and sends output to files with the extension **.i**.
-s	Strips the entire table, leaving machine instructions only.
-S	Creates an assembly source listing. Suppresses creation of an executable file.
-V*string*	Copies *string* to the object file.
-w	Suppresses compiler warning messages.
-W *num*	Sets the output level for compiler warning messages.
-X	Removes the standard directories from the list of directories where **cc** searches for **#include** files.

Memory Models

The following table defines the number of text and data segments for the four program memory models. This table also lists the segment register values.

Model	**Text**	**Data**	**Segment Registers**
Small	1*	1*	CS=DS=SS (impure)
Middle	1 or more**	1	DS=SS
Large	1 or more**	1 or more	DS=SS
Huge	1 or more**	1 or more	DS=SS

* In impure-text small module programs, text and data occupy the same segment. In pure-text programs, they occupy different segments.

** A single module cannot use more than one text segment, but the entire program can use more than one text segment.

Pointer and Integer Sizes

The following table defines the sizes (in bits) of integers (**int** type) and text and data pointers in each program memory model.

Model	**Data Pointer**	**Text Pointer**	**Integer**
Small	16	16	16
Middle	16	32	16
Large	32	32	16
Huge	32	32	16

Segment and Module Names

The following table lists the default text and data segment names and the default module name for each object file.

Model	**Text**	**Data**	**Module**
Small	_TEXT	_DATA	*filename*
Middle	*module*_TEXT	_DATA	*filename*
Large	*module*_TEXT	_DATA	*filename*
Huge	*module*_TEXT	_DATA	*filename*

Chapter 2. Include Files

The following two groups contain **include** files:

/usr/include	These are useful to individual users.
/usr/include/sys	These reference system information.

You can, however, also use a number of the **include** files in the **sys** directory.

This chapter contains summary descriptions of the PC XENIX **include** files. These descriptions are divided into two sections.

/usr/include Files

This section describes the function of each standard **include** file. It also lists the routines you find in each. These **include** files may also contain macro and constant definitions, type definitions, function declarations, and structure definitions.

Notes show where these declarations or definitions are of special interest. For detailed information on a particular routine, see the appropriate manual page in *XENIX System Reference*.

ar.h	A header, which the structure **ar_hdr** determines, precedes each file placed in an archive file. The **ar.h** file also sets the value of the archive magic number.
assert.h	Defines a macro that is useful in verifying the validity of a specified C statement. See **assert** for more information.
core.h	Defines the location and size of a core-image file. See **core** for detailed information on the structure of core files.
ctype.h	Defines a number of macros that classify ASCII-coded integer values by doing a table lookup. For a complete list of the available macros, see **ctype**.

curses.h	Provides a number of routines that control screen and cursor functions. See **curses** for a complete list of all the functions available.
dbm.h	Defines the functions: **dbminit** **fetch** **nextkey** **delete** **firstkey** **store** to work with databases. These routines can handle very large (1 billion[1] blocks) databases. See **dbm** for more detailed information.
dial.h	Defines the routines **dial** and **undial**. You can use these in communication over phone lines between PC XENIX and UNIX[2] systems. It also defines the structure CALL, which stores information about the communication attempt.
dumprestor.h	When you do incremental dumps onto magnetic tapes, information defined by the structure **spcl** precedes the files that you are dumping. This structure defines the format of the header record and the first record of each description. The structure **idates** describes an entry to the file that keeps the dump history.
errno.h	This file contains definitions of error codes that the kernel passes to the external variable **errno**. When an error condition occurs during a system call, the kernel sets the **errno** variable to the appropriate value. See **perror** for information on error handling.
execargs.h	Declares and initializes external pointer **execargs**.
fcntl.h	Provides the values for the file control function, **fcntl**. For a description of the values, see **fcntl**.
ftw.h	Contains codes for the third argument to the user-supplied function that you name as the second argument to **ftw**.

---

[1] One billion equals 10^9.

[2] Trademark of AT&T Bell Laboratories.

grp.h Defines a structure **group**, which returns pointers to information about entries in the file **/etc/group**. See **getgrent** for more information.

macros.h Defines a number of useful macros (some for string handling, others for library routines).

malloc.h Defines the **mallinfo** structure, which contains information on memory allocation. Defines the routines:

malloc **free** **realloc**
mallopt **mallinfo**

See **malloc** for more information.

math.h Contains **#defines** for the following math routines:

atof **frexp** **ldexp** **modf**
erf **erfc** **exp** **log**
log10 **pow** **sqrt** **floor**
ceil **fmod** **fabs** **gamma**
hypot **matherr** **sinh** **cosh**
tanh **sin** **cos** **tan**
acos **atan** **atan2** **j0**
j1 **jn** **y0** **y1**
yn **asin**

It also defines a number of useful mathematical constants. See **bessel**, **exp**, **floor**, **hypot**, **gamma**, **sinh**, and **trig** for detailed information on the math functions.

memory.h Defines the routines:

memccpy **memchr** **memcmp**
memcpy **memset** **movedata**

These routines manipulate buffers.

mnttab.h The structure **mnttab** defines the format of the **/etc/mnttab** file. This file keeps a record of special files mounted using the **mount** command. See **mount** for more information.

mon.h Defines two structures: **monhdr** and **mon**. These structures determine the format of the buffer in which **monitor** stores information on the

	execution profile of a specified program. For more information, see **monitor** and **profil**.
pwd.h	Defines two structures: **passwd** and **comment**. These structures determine the format of the entries in the **/etc/passwd** file. They also determine the format of the comments associated with these entries. See **getpwent** for details on the structure of the entries.
regexp.h	Defines the following functions: **compile** **step** **advance** **getrnge** **ecmp** These functions compile XENIX regular expressions and return pointers to the compiled forms. For a detailed description, see **regex**.
sd.h	Defines a number of flags for the **sdget** system call. Defines the **sdget** system call. For more information, see **sdget**.
search.h	Defines types used in tree searches.
setjmp.h	Provides data to insure that the **setjmp** and **longjmp** system calls are machine-independent.
sgtty.h	Defines the structure **sgttyb** for the **stty** and **gtty** system calls. It also defines the **stty** and **gtty** system calls, terminal modes, delay algorithms, speeds, and **ioctl** arguments. Additionally, it defines the structure **tchars**, for dealing with special characters. For more information, see **ioctl**, **tty**, and **stty**.
signal.h	Defines the values that the kernel can assign to signal. These values return to the calling process upon receipt of an error. For more detail, see **signal**.
stand.h	Provides the necessary information and structures for the operation of the system in standalone mode.

stdio.h	Defines the standard buffered input and output routines, and the files: **stdin**, **stdout**, and **stderr**. It also defines the following macros: **getchar** **putchar** **ftell** **rewind** **setbuf** Defines macros for: **clearerr**, **feof**, **ferror**, and **fileno**.
string.h	Defines the following string manipulation routines: **strcmp** **strncmp** **strlen** **strspn** **strcspn** For details, see **string**.
termio.h	Defines characters and modes for the terminal interface. In addition, it defines a structure for the **ioctl** system calls. For more information, see **ioctl** and **tty**.
time.h	Defines the structure for the conversion of time to ASCII. Defines the routine **tzset** and the variables: **timezone**, **daylight**, and **tzname**. See **ctime** for details.
unlstd.h	Defines the flag values for the **lock** system call. See **lock** for details.
ustat.h	Defines the structure **ustat**, which returns information about a given mounted file system. For details, see **ustat**.
utmp.h	Defines the format for the **/etc/utmp** system accounting file. See **utmp** for details.
values.h	Defines various values for machine-dependent variables.
varargs.h	Contains macros for use in variable argument functions. Provided to allow portability of C language code.

/usr/include/sys Files

The following **include** files are system files. Many of them define system parameters, or contain information the kernel uses. For detailed information on a particular routine, see the appropriate page in the indicated section of the *XENIX System Reference.*

a.out.h	Declares the following structures: **xexec** (defines the **x.out** header) **xext** (defines the **x.out** header extension) **xseg** (defines the **x.out** segment table entry) **xiter** (defines the **x.out** iteration record) **xlist** (defines the structure for the **xlist** system call) **aexec** (defines the **a.out** header) **nlist** (defines the structure for the **nlist** system call) **bexec** (defines the **b.out** header). See **a.out** for more detailed information.
acct.h	Defines the structure **acct** for use in system accounting. See **acct** and **accton** for more information.
assert.h	Defines the ASSERT macro. See **assert** for more details.
brk.h	Defines the commands for break control.
buf.h	Defines the structures: **buf** and **hbuf**. These structures (respectively) provide access to an I/O buffer for device drivers, and fast access to the buffers through hashing.
callo.h	Defines the structure **callo**, which allows a clock interrupt for a specific period. The **timeout** function uses this structure.
conf.h	Defines the structures: **linesw**, **bdevsw**, and **cdevsw**. These structures are the declarations (respectively) for the line discipline switch, the block device switch and the character device switch.
dir.h	Defines the structure **direct**, which contains the value for the maximum directory size.

errno.h	Contains the values for the **errno** variable. The kernel sets the **errno** variable upon an error. Math routines also set it. See **perror** for more information.
fblk.h	Defines the structure **fblk**, which contains the address of the next free block.
file.h	Defines a structure **file**, which holds the read/write pointer associated with each open file.
filsys.h	Defines the structure of the super-block and a number of fundamental system variables.
ino.h	Defines the structure of the inode as it appears on a disk block.
inode.h	Contains definitions of the structures: **iisem**, **iisd**, and **inode**. These structures (respectively) provide information about the semaphores related to a given inode, the shared data segments allocated to the inode, and the inode itself.
iobuf.h	Defines the structure of the I/O buffer for each block device.
ioctl.h	Defines macros for I/O control.
ipc.h	Provides constant definitions for the inter-process communications (IPC) report. See **ipc** for more information.
lock.h	Defines flag values for the locking of resources.
locking.h	Defines flag values for the **locking** system call. Defines the structure **locklist**, which provides the structure for the linked list of lock regions.
machdep.h	Defines machine-dependent variables (for example, number of descriptor table entries or clock timing).
map.h	Defines the structure **map** to hold the location of the swapmap.
mmu.h	Defines constants for the descriptor tables for memory management purposes.
mount.h	Defines the structure **mount**. PC XENIX allocates one for every device mounted. See **mount** for more information.

msg.h

Defines the structures:

msqid_ds (The data structure for interprocess message queues)
msg (A structure for each message in a queue)
msgbuf (The user message buffer template for the **msgsnd** and **msgrecv** system calls)
msqinfo (A structure containing information about the state of the message queue).

See **ipcs**, **msgctl**, **msgget**, and **msgop** for more information on interprocess communication.

param.h

Contains the following parameters that are vital to the system:

- Adjustable parameters
- Priorities
- Signals
- MMU constants
- Macros for unit conversion
- Definitions of the fundamental constants of the implementation.

proc.h

Defines the structures **proc**, and **xproc**. These structures hold all the vital information on processes. You can swap them out.

reg.h

Defines constants that provide an index of the available registers relative to AX.

relsym.h

Provides definitions for the following structures:

sym
asym
bsym
reloc
xreloc

The structure **sym** defines the structure of the symbol table for **x.out**. The structure **reloc** defines a relocation table entry for the long form of **x.out**, and **xreloc** defines the relocation table entry for the short form of **x.out**. The structures **asym** and **bsym** provide compatibility with other systems.

The **relsym.h** file also provides definitions for:

sym.s_type
asym.sa_type
reloc.r_desc
nlist.n_type

relsym86.h	Contains the declarations for the 8086/ 80286 symbol table and relocation record structures. Defines the structures: **dosexec**, **desctab**, and **srel86**. The structure **dosexec** provides for DOS support, **desctab** provides the structure of the descriptor table, and **srel86** provides the structure for segment relocation. Segment relocation is necessary for middle and larger model memory support.
sd.h	Defines the shared data table. See the following reference to **sdu.h**.
sdu.h	Defines values for the shared data flags that the following shared data system calls use: **sdgetv** **sdget** **sdenter** **sdleave** **sdfree** **sdwaitv** See the appropriate pages in *XENIX System Reference* for more information.
sem.h	Defines the structures used by the semaphore operations system call, **semop**. The structures are: **semid_ds** **sem** **sem_undo** **seminfo** **sembuf** See **semop** for detailed information.
signal.h	Defines the values for the SIGNAL constants. See **signal** for more information.
sites.h	Provides values for system constants that the structure defined in **utsname.h** uses.
space.h	Defines global data structures for the kernel. These include: the previous entry table, the inode table, the buffers, and assorted variables that PC XENIX uses.
stat.h	Defines the structure **stat**, which returns a structure to both the **stat** and **fstat** system calls. Also defines a number of constants.
sysinfo.h	Defines the structures, **sysinfo** and **syswait**, which hold information about the state of the system and its processes.

sysmacros.h	Defines a number of machine-dependent macros.
systm.h	Defines the structures **sysent**, and **idt**, which define the format for the system-entry table and the interrupt descriptor table. Also defines a number of random variables and functions used by more than one routine.
text.h	Defines the structure **text**, which provides the format for text segments. Also defines a number of constants.
timeb.h	Defines the structure **timeb**, returned by the **ftime** system call. See **time** for more information.
times.h	Defines the structure **tms**, returned by the routine **times()**. See **times** for more information.
ttold.h	Defines the structures **sgtty** and **tc**, which contain information for the **stty** and **gtty** system calls. Also defines the terminal modes.
tty.h	Defines the structures: **tty** **clist** **cblock** **chead** **inter** The **tty** structure formats the information for I/O for each character device. The **tty.h** file also defines a number of internal state variables and device commands.
types.h	Defines the structure **saddr** and numerous machine-dependent variables.
ulimit.h	Defines values passed to the **ulimit** system call.
user.h	Defines the structure **user**, which contains all the data on a user process that PC XENIX does not need to reference. PC XENIX swaps this with the process. The **user.h** file also redefines the standard error codes.
utsname.h	Defines the structure **utsname**, which provides general information about system characteristics.
var.h	Defines the structure **var**.

Chapter 3. Using the Standard I/O Functions

This chapter explains how to use the I/O functions in the standard C library. In particular, it describes:

- Command line arguments
- Standard input and output files
- Stream functions for ordinary files
- Low-level functions for ordinary files
- Random access functions.

Preparing for the I/O Functions

To use the standard I/O functions, a program must include the file **stdio.h**. This file defines the needed macros and variables. To include this file, place the following line at the beginning of the program:

```
#include <stdio.h>
```

The library file **libc.a** contains the functions. The compiler reads this file whenever you compile a program. You do not need a special argument when you invoke the compiler.

Special Names

The standard I/O library uses many names for special purposes. In general, you can use these names in any program that has included the **stdio.h** file. The special names are:

stdin	The name of the standard input file.
stdout	The name of the standard output file.
stderr	The name of the standard error file.

EOF	The value returned by the read routines on end-of-file or error.
NULL	The null pointer, returned by pointer-valued functions, to indicate an error.
FILE	The name of the file type used to declare pointers to streams.
BUFSIZ	The size in bytes suitable for an I/O buffer supplied by the user.

Special Macros

The functions **getc**, **getchar**, **putc**, **putchar**, **feof**, **ferror**, and **fileno** are actually macros, not functions. This means that you cannot redeclare them or use them as targets for a breakpoint when debugging.

Using Command Line Arguments

PC XENIX lets you pass information to a program at the same time you invoke the program for execution. You can do this with command line arguments.

A command line is the line you type to invoke a program. A command line argument is anything you type in a command line. A command line argument can be a file name, an option, or a number. The first argument in any command line must be the file name of the program you wish to execute.

When you type a command line, the system reads the first argument and loads the corresponding program. It also counts the other arguments and stores them in memory in the same order in which they appear on the line. Then it passes the count and the locations to the main function of the program. The function can then access the arguments by accessing the memory in which they are stored.

To access the arguments, the main function must have two parameters: **argc**, an integer variable containing the argument count, and **argv**, an array of pointers to the argument values. You can define the parameters by using the lines:

```
main(argc, argv)
int argc;
char *argv[ ];
```

at the beginning of the main program function. When a program begins execution, **argc** contains the number of arguments, and each element in **argv** contains a pointer to one argument.

An argument is stored as a null-terminated string, that is, a string ending with a null character (\0). The first string (at **argv**[0]) is the program name. The argument count is never less than 1, because the program name is always the first argument.

In the following example, the compiler reads command line arguments and echoes them on the terminal screen. This program is similar to the **echo** command:

```
main(argc, argv)          /* echo arguments */
int argc;
char *argv[ ];
{
   int i;

   for (i = 1; i < argc; i++)
      printf("%s%c", argv[i], (i < argc - 1) ? ' ' : '\n');
}
```

In this example, an extra space character is added at the end of each argument to separate it from the next argument. You must do this, because the compiler removes leading and trailing white space characters (that is, spaces and tabs) when it reads the arguments from the command line.

When typing arguments on a command line, separate each argument from the others by one or more whitespace characters. If an argument must contain whitespace characters, enclose that argument in double quotation marks. For example, in the command line:

```
display 3 4 "echo hello"
```

the string “echo hello” is a single argument. Also enclose in double quotation marks any argument that contains characters recognized by the shell, such as <, >, ¦, and ^.

You should not change the values of the **argc** and **argv** variables. If necessary, assign the argument value to another variable and change that variable instead. You can give other functions in the program access to the arguments by assigning their values to external variables.

Using the Standard Files

Whenever you invoke a program for execution, PC XENIX creates a standard input, a standard output, and a standard error file to handle the input and output needs of the program. The bulk of input and output of most programs is through your own terminal. The system, therefore, assigns your terminal keyboard and screen as the standard input and output, respectively. It also assigns the standard error file, which receives any error messages generated by the program, to your screen.

A program can read and write to the standard input and output files with the **getchar**, **gets**, **scanf**, **putchar**, **puts**, and **printf** functions. You can access the standard error file with the stream functions described in "Using the Stream Functions" on page 3-12.

PC XENIX lets you redirect the standard input and output using the shell's redirection symbols (< and >). This allows a program to use other devices and files as its chief source of input and output.

The following sections explain how to read from and write to the standard input and output. They also explain how to redirect the standard input and output.

Reading from the Standard Input

You can read from the standard input with the **getchar**, **gets**, and **scanf** functions.

The **getchar** function reads one character at a time from the standard input. The function call has the form:

```
c = getchar()
```

where *c* is the variable to receive the character. *c* must have **int** type. The function normally returns the character read. It returns the end-of-file value, **EOF**, if it encounters the end of the file or an error.

In a conditional loop, the **getchar** function reads a string of characters from the standard input. For example, the following function reads *cnt* number of characters from the keyboard:

```
#include <stdio.h>

readn(p, cnt)
char p[ ];
int cnt;
{
   int c;
   int i = 0;

   while (i < cnt  &&  (c = getchar()) != EOF)
      p[i++] = c;
   return(i);
}
```

If **getchar** is reading from the keyboard, it waits for you to type characters before returning.

The **gets** function reads a string of characters from the standard input and copies the string to a given memory location. The function call has the form:

gets(*s*)

where *s* is a pointer to the location to receive the string. The function reads characters until it finds a new-line character (\n). It then replaces the new-line character with a null character (\0) and copies the resulting string to memory. The function returns the null pointer value **NULL** if it encounters the end of the file or an error. Otherwise, it returns the value of *s*.

The function reads a full line from the standard input. For example, the following program fragment reads a line from the standard input, stores it in the character array **cmdln**, and calls a function (called **parse**) if no error occurs:

```
char cmdln[SIZE];

if (gets(cmdln) != NULL)
   parse();
```

In this case, the length of the string must be less than **SIZE**. The **gets** function cannot check the length of the string; therefore, an overflow can occur.

The **scanf** function reads one or more values from the standard input where a value may be a character string or a decimal, octal, or hexadecimal number. The function call has the form:

scanf(*format*, *argptr* . . .)

where *format* is a pointer to a string that defines the format of the values to be read and *argptr* is one or more pointers to the variables that receive the values. There must be one *argptr* for each format given in the *format* string. (The * format is an exception to this rule.) The format may be %s for a string, %c for a character, and %d, %o, or %x for a decimal, octal, or hexadecimal number, respectively. Other formats, including the * format, are described in **scanf** in *XENIX System Reference.*) The function normally returns the number of values it reads from the standard input. It returns the value **EOF** if it encounters the end of the file or an error.

Unlike the **getchar** and **gets** functions, **scanf** skips all whitespace characters, reading only those characters that make up a value. It then converts the characters, if necessary, into the appropriate string or number.

Use the **scanf** function whenever you need formatted input, that is, input that you must type in a special way or that has a special meaning. For example, in the following program fragment, **scanf** reads both a name and a number from the same line:

```
char name[20];
int number;

scanf("%s %d", name, &number);
```

In this example, the string "%s %d" defines the values that are to be read (a string and a decimal number). The string is copied to the character array *name* and the number to the integer variable *number*. We use pointers to these variables in the call and not the actual variables themselves.

When reading from the keyboard, **scanf** waits for you to type values before returning. Separate each value from the next by one or more white space characters, such as spaces, tabs, or even new-line characters. For example, for the function:

```
scanf("%s %d %c", name, &age, &sex);
```

an acceptable input is:

```
Jeni 22
F
```

If a value is a number, it must have the appropriate digits, that is, a decimal number must have decimal digits, octal numbers octal digits, and hexadecimal numbers hexadecimal digits.

If **scanf** detects an error, it immediately stops reading the standard input. Before you can use **scanf** again, you must use the **getchar** function to remove the illegal character that caused the error from the input.

You may use the **getchar**, **gets**, and **scanf** functions in a single program. Just remember that each function reads the next available character, making that character unavailable to the other functions.

When the standard input is the terminal keyboard, the **getchar**, **gets**, and **scanf** functions usually do not return a value until you have typed at least one new-line character. This is true even if you want only one character. If you want to have immediate input on a single keystroke, see the example under "Using the Standard Screen" on page 4-13.

Writing to the Standard Output

You can write to the standard output with the **putchar**, **puts**, and **printf** functions.

The **putchar** function writes a single character to the output buffer. The function call has the form:

```
putchar(c)
```

where *c* is the character you want to write. The function normally returns the same character it wrote but returns the value **EOF** if it encounters an error.

The function in a **for** or **while** loop writes a string of characters to the standard output. For example, the function:

```
#include <stdio.h>

write(p,cnt)
char p[ ];
int cnt;
{
   int i;

   for (i = 0; i < cnt; i++)
      putchar(p[i]);
   putchar('\n');
}
```

writes *cnt* number of characters plus a new-line character to the standard output.

The **puts** function copies the string found at a given memory location to the standard output. The function call has the form:

`puts(`*`s`*`)`

where *s* is a pointer to the location containing the string. The string can be any number of characters but must end with a null character (\0). The function writes each character in the string to the standard output. The function replaces the null character at the end of the string with a new-line character.

Because the function appends a new-line character, use it when writing full lines to the standard output. For example, the following program fragment writes one of three strings to the standard output:

```
int c;


switch (c)
{
    case('1'):
        puts("Continuing . . . ");
        break;
    case('2'):
        puts("All done.");
        break;
    default:
        puts("Sorry, there was an error.");
}
```

The string that the above program writes depends on the value of *c*.

The **printf** function writes one or more values to the standard output where a value is a character string or a decimal, octal, or hexadecimal number. The function converts numbers into the proper display format. The function call has the form:

`printf(`*`format`*`[, `*`arg`*`] . . . )`

where *format* is a pointer to a string that describes the format of each value the function writes and *arg* is one or more variables containing the values the function writes. There must be one *arg* for each format in the *format* string. The formats may be %s for a string, %c for a character, and %d, %o, or %x for a decimal, octal, or hexadecimal number, respectively. (Other formats are described under **printf** in *XENIX System Reference*.) If you request a string, the corresponding *arg* must be a pointer. The function returns the number of characters placed, unless it encounters an error. If it encounters an error, the function returns a negative value.

Use the **printf** function when you need formatted output, that is, when the output must appear a certain way. For example, you may use the function to display a name and number on the same line as in the following example:

```
char name [ ];
int number;

printf("%s %d", name, number);
```

In this example, the string "%s %d" defines the type of output the function displays (a string and a number separated by a space). The output values are copied from the character array name and the integer variable number.

You may use the **putchar**, **puts**, and **printf** functions in a single program. Just remember that the output appears in the same order as it is written to the standard output.

Redirecting the Standard Input

You can change the standard input from the terminal keyboard to an ordinary file by using the normal shell redirection symbol, <. This symbol directs the shell to open for reading the file whose name immediately follows the symbol. For example, the following command line opens the file **phonelist** as the standard input to the program **dial**:

```
dial <phonelist
```

The **dial** program may then use the **getchar**, **gets**, and **scanf** functions to read characters and values from this file. If the file does not exist, the shell displays an error message and stops the program.

Whenever **getchar**, **gets**, or **scanf** read from an ordinary file, they return the value **EOF** if they encountered the end of the file or an error. It is useful to check for this value to make sure you do not continue to read characters after an error has occurred.

Redirecting the Standard Output

You can change the standard output of a program from the terminal screen to an ordinary file by using the shell redirection symbol, `>`. The symbol directs the shell to open for writing the file whose name immediately follows the symbol. For example, the command line:

```
dial >savephone
```

opens the file **savephone**, not the terminal screen, as the standard output of the program **dial**. You may use the **putchar**, **puts**, and **printf** functions to write to the file.

If the file does not exist, the shell automatically creates it. If the file exists, but the program does not have permission to change or alter the file, the shell displays an error message and does not execute the program.

Piping the Standard Input and Output

Another way to redefine the standard input and output is to create a pipe. A pipe simply connects the standard output of one program to the standard input of another. The programs may then use the standard input and output to pass information from one to the other. You can create a pipe by using the standard shell pipe symbol, ¦.

For example, the command line:

```
dial ¦ wc
```

connects the standard output of the program **dial** to the standard input of the program **wc**. (This command line does not affect the standard input of **dial** and the standard output of **wc**.) If **dial** writes to its standard output with the **putchar**, **puts**, or **printf** functions, **wc** can read this output with the **getchar** and **scanf** functions.

When the program on the output side of a pipe terminates, the system automatically places the constant value **EOF** in the standard input of the program on the input side. Pipes are described in more detail in Chapter 7, "Creating and Using Pipes."

Program Example

This section shows how to use the standard input and output files to perform useful tasks. The **ccstrip** (for control character strip) program defined below strips out all ASCII control characters from its input except for the new-line character and the tab character. This program displays text or data files that contain characters that can disrupt your terminal screen.

```
#include <stdio.h>
#include <ctype.h>

main()  /* ccstrip: print only visible
           ASCII, tab or new-line characters        */
{
    int c;

    for (;;)
    {
        if ((c = getchar()) == EOF) /* end-of-file? */
            if (feof(stdin))
                break;

        if (!iscntrl(c) && (c < 0x7f))
            putchar(c);
    }
}
```

You can strip and display the contents of a single file by changing the standard input of the **ccstrip** program to the desired file. The command line:

```
ccstrip <doc.t
```

reads the contents of the file **doc.t**, strips out control characters, then writes the stripped file to the standard output.

To strip several files at the same time, you can create a pipe between the **cat** command and **ccstrip**.

To read and strip the contents of the files **file1**, **file2**, and **file3**, then display them on the standard output, use the command:

```
cat file1 file2 file3 ¦ ccstrip
```

To save the stripped files, redirect the standard output of **ccstrip**. For example, this command line writes the stripped files to the file **clean**:

```
cat file1 file2 file3 ¦ ccstrip >clean
```

The **exit** function at the end of the program ensures that any program that executes the **ccstrip** program receives a normal termination status (0) from the program when it ends. An explanation of the **exit** function and how to execute one program under control of another is given in “Stopping a Program” on page 6-3.

Using the Stream Functions

The functions described so far all read from the standard input and write to the standard output. The next step is to show functions that access files not already connected to the program. One set of standard I/O functions allows a program to open and access ordinary files as if they were a stream of characters. For this reason, the functions are called the stream functions.

Unlike the standard input and output files, you must use the **fopen** function to open a file that a stream function accesses. The function can open a file for reading, writing, or appending. A program can read from a file with the **getc**, **fgetc**, **fgets**, **fgetw**, **fread**, and **fscanf** functions. It can write to a file with the **putc**, **fputc**, **fputs**, **fputw**, **fwrite**, and **fprintf** functions. A program can test for the end of the file or for an error with the **feof** and **ferror** functions. A program can close a file with the **fclose** function.

Using File Pointers

Every file opened for access by the stream functions has a unique pointer associated with it called a file pointer. This pointer, defined with the predefined type **FILE** found in the **stdio.h** file, points to a structure that contains information about the file, such as the location of the buffer (the intermediate storage area between the file and the program), the current character position in the buffer, and whether the file is being read or written. You can give the pointer a valid pointer value with the **fopen** function as described in the next section. (The **NULL** value, like **FILE**, is defined in the **stdio.h** file.) Thereafter, the file pointer refers to that file until you explicitly close the file with the **fclose** function.

Following is an example file pointer definition:

```
FILE   *infile;
```

The standard input, output, and error files, like other opened files, have corresponding file pointers. The names of the file pointers are **stdin** for standard input, **stdout** for standard output, and **stderr** for standard error. Unlike other file pointers, the standard file pointers are predefined in the **stdio.h** file. This means a program uses these pointers to read and write from the standard files without first using the **fopen** function to open them.

The predefined file pointers are used when a program needs to alternate between the standard input or output file and an ordinary file. Although the predefined file pointers have **FILE** type, they are constants, not variables. You must not assign them values.

Opening a File

The **fopen** function opens a given file and returns a file pointer to a structure containing the data necessary to access the file. The pointer, in subsequent stream functions, reads from or writes to the file.

The function call has the form:

fp = `fopen(`*filename*`,` *type*`)`

where *fp* is the pointer to receive the file pointer, *filename* is a pointer to the name of the file you want to open and *type* is a pointer to a string that defines how the file is to be opened. The type string may be **r** for reading, **w** for writing, or **a** for appending, that is, open for writing at the end of the file.

You can open a file for different operations at the same time if you use separate file pointers. For example, the following program fragment opens the file named **/usr/accounts** for both reading and writing:

```
FILE *rp, *wp;

rp = fopen("/usr/accounts", "r");
wp = fopen("/usr/accounts", "a");
```

Opening an existing file for writing destroys the old contents. Opening an existing file for appending leaves the old contents unchanged and causes any data written to the file to be added to the end.

Trying to open a nonexistent file for reading causes an error. Trying to open a nonexistent file for writing or appending causes a new file to be created. Trying to open any file for which the program does not have appropriate permission causes an error.

The **fopen** function normally returns a valid file pointer, but it returns the value **NULL** if it encounters an error upon opening the file. Check for the **NULL** value after each call to the function to prevent reading or writing after an error.

Reading a Single Character

The **getc** and **fgetc** functions return a single character read from a given file, and return the value **EOF** if they encounter the end of the file or an error. The function calls have the forms:

c = `getc(`*stream*`)`

and

c = `fgetc(`*stream*`)`

where *stream* is the file pointer to the file to be read and *c* is the variable to receive the character. The return value is always an integer.

Use the functions in conditional loops to read a string of characters from a file. For example, the following program fragment continues to read characters from the file given to it by **infile** until it encounters the end of the file or an error:

```
int c, i = 0;
char buf[MAX];
FILE *infile;

for (;;)
{
   if ((c = getc(infile)) == EOF)
      if (feof(infile))
         break;

   if (i < MAX)
      buf[i++] = c;
   else
      break;
}
```

The only difference between the functions is that **getc** is defined as a macro, and **fgetc** is defined as a true function. This means that unlike **getc**, you can:

- Pass **fgetc** as an argument in another function.
- Use **fgetc** as a target for a breakpoint when debugging.
- Use **fgetc** to avoid any side effects of macro processing.

Reading a String from a File

The **fgets** function reads a string of characters from a file and copies the string to a given memory location. The function call has the form:

`fgets(`*`s`*`, `*`n`*`, `*`stream`*`)`

where *s* is a pointer to the location to receive the string, *n* is a count of the maximum number of characters to be in the string, and *stream* is the file pointer of the file to be read. The function reads *n*-1 characters or up to the first new-line character, whichever occurs first. The function appends a null character (\0) to the last character read and then stores the string at the specified location. The function returns the null pointer value **NULL** if it encounters the end of the file or an error. Otherwise, it returns the pointer *s*.

Use this function to read a full line from a file. For example, the following program fragment reads a string of characters from the file given by **myfile**:

```
char cmdln[MAX];
FILE *myfile;

if (fgets(cmdln, MAX, myfile) != NULL)
   parse(cmdln);
```

In this example, **fgets** copies the string to the character array **cmdln**.

Reading Records from a File

The **fread** function reads one or more records from a file and copies them to a given memory location. The function call has the form:

`fread(`*`ptr`*`, `*`size`*`, `*`nitems`*`, `*`stream`*`)`

where *ptr* is a pointer to the location to receive the records, *size* is the size (in bytes) of each record to be read, *nitems* is the number of records to be read, and *stream* is the file pointer of the file to be read. The *ptr* may be a pointer to a variable of any type (from a single character to a structure). The *size*, an integer, should give the numbers of bytes in each item you wish to read. One way to ensure this is to use the **sizeof** function on the pointer *ptr* (see the following example). The function always returns the number of items it read, regardless of whether it encounters the end of the file or an error.

Use this function to read binary data from a file. For example, the following program fragment reads two records from the file given by **database** and copies the records into the structure **person**:

```
#include <stdio.h>

main()
{
   FILE *database;
   struct record
   {
      char name[5];
   } person[3];

   if ((database = fopen("fwfile", "r")) == NULL)
   {
      fprintf(stderr, "cannot open fwfile\n");
      exit(1);
   }
   if (fread(person, sizeof(struct record), 2, database) != 2)
      exit(1);
}
```

The **fread** function does not explicitly indicate errors, so use the **feof** and **ferror** functions to detect end of the file and errors. (See "Testing for the End of a File" on page 3-20 and "Testing for File Errors" on page 3-21.)

Reading Formatted Data from a File

The **fscanf** function reads formatted input from a given file and copies it to the memory location given by the respective argument pointers, in the same way as the **scanf** function reads from the standard input. The function call has the form:

`fscanf(`*stream*`,` *format*`,` *argptr* `. . . )`

where *stream* is the file pointer of the file to be read, *format* is a pointer to the string that defines the format of the input to be read, and *argptr* is one or more pointers to the variables that are to receive the formatted input. There must be one *argptr* for each format given in the *format* string. The format may be %s for a string, %c for a character, and %d, %o, or %x for a decimal, octal, or hexadecimal number, respectively. (Other formats are described under **scanf** in *XENIX System Reference.*) The function normally returns the number of arguments it read but returns the value **EOF** if it encounters the end of the file or an error.

The function reads files that contain both numbers and text. For example, this program fragment reads a name and a decimal number from the file given by **file**:

```
FILE *pfile;
int pay;
char name[20];

fscanf(pfile, "%s %d\n", name, &pay);
```

This program fragment copies the name to the character array **name** and the number to the integer variable **pay**.

Writing a Single Character

The **putc** and **fputc** functions write single characters to a given file. The function calls have the forms:

`putc(`*c*`,`*stream*`)`

and

`fputc(`*c*`,`*stream*`)`

where *c* is the character to be written and *stream* is the file pointer to the file to receive the character. The function normally returns the character written, but returns the value **EOF** if it encounters an error.

The **putc** function is a macro and may have undesirable side effects such as function calls or increment or decrement operators resulting from argument processing. In such cases, use the equivalent function **fputc**.

Use these functions in conditional loops to write a string of characters to a file. For example, the following program fragment writes characters from the array name to the file given by **out**:

```
FILE *out;
char name[MAX];
int i;

for (i=0; i<MAX; i++)
   fputc(name[i], out);
```

The only difference between the **putc** and **fputc** functions is that **putc** is a macro and **fputc** is an actual function. This means that you can use **fputc**, unlike **putc**, as an argument to another function, as the target of a breakpoint when debugging, and to avoid the restriction of macro processing. (This restriction is that the arguments of macro functions should not contain any side effects such as function calls or increment or decrement operators.)

Writing a String to a File

The **fputs** function writes a string to a given file. The function call has the form:

`fputs(`*s*`,`*stream*`)`

where *s* is a pointer to the string to be written and *stream* is the file pointer to the file.

The function copies strings from one file to another. For example, in the following program fragment, **gets** and **fputs** copy strings from the standard input to the file given by **out**:

```
FILE *out;
char cmdln[MAX];

if (gets(cmdln) != NULL)
   fputs(cmdln, out);
```

The function normally returns zero but returns **NULL** if it encounters an error.

Writing Formatted Output

The **fprintf** function writes formatted output to a given file, in the same way the **printf** function writes to the standard output. The function call has the form:

`fprintf(`*stream*`,` *format*`, [`*arg*`] . . . )`

where *stream* is the file pointer of the file to be written to, *format* is a pointer to a string that defines the format of the output, and *arg* is one or more arguments to be written. There must be one *arg* for each format in the *format* string. The formats may be %s for a string, %c for a character, and %d, %o, or %x for a decimal, octal, or hexadecimal number, respectively. (Other formats are described under **printf** in *XENIX System Reference.*) If you request a string, the corresponding *arg* must be a pointer; otherwise, you must use the actual variable. The function normally returns zero, but returns a nonzero number if it encounters an error.

The **fprintf** function writes output that contains both numbers and text. For example, to write a name and a decimal number to the file given by **outfile**, use the following program fragment:

```
FILE *outfile;
int pay;
char name[20];

fprintf(outfile, "%s%d\n", name, pay);
```

The name is copied from the character array **name** and the number from the integer variable **pay**.

Writing Records to a File

The **fwrite** function writes one or more records to a given file. The function call has the form:

`fwrite(`*ptr*`,` *size*`,` *nitems*`,` *stream*`)`

where *ptr* is a pointer to the first record to be written, *size* is the size (in bytes) of each record, *nitems* is the number of records to be written, and *stream* is the file pointer of the file. The *ptr* may point to a variable of any type (from a single character to a structure). The *size* should give the number of bytes in each item to be written. One way to ensure this is to use the **sizeof** function (see the following example). The function always returns the number of items actually written to the file, whether it encounters the end of the file or an error.

The **fwrite** function writes binary data to a file. For example, the following program fragment writes two records to the file given by **database**:

```
#include <stdio.h>

struct record
{
    char name[5];
} person[3] =
{
   'M', 'i', 'k', 'e', ' ',
   'T', 'o', 'm', ' ', ' ',
   'B', 'i', 'l', 'l', ' '
};

main()
{
   FILE *database;
   if ((database = fopen("fwfile", "w")) == NULL)
   {
      fprintf(stderr, "cannot open fwfile\n");
      exit(1);
   }
   fwrite(person, sizeof(struct record), 2, database);
}
```

The records are copied from the structure **person**.

Because the function does not report the end of the file or errors, use the **feof** and **ferror** functions to detect these conditions.

Testing for the End of a File

The **feof** function returns a nonzero value if a given file has reached its end. The function call has the form:

`feof(`*stream*`)`

where *stream* is the file pointer of the file. The function returns a nonzero value only if the file has reached its end; otherwise, it returns 0. The return value is always an integer.

Use the **feof** function after functions whose return value is not a clear indicator of an end-of-file condition. For example, in the following program fragment, the function checks for the end of the file after each 10 characters are read. The reading stops as soon as **feof** returns -1.

```
char name[10];
FILE *stream;

while (!feof(stream))
   fread(name, sizeof(name), 1, stream);
```

Testing for File Errors

The **ferror** function tests a given stream file for an error. The function call has the form:

`ferror(`*stream*`)`

where *stream* is the file pointer of the file to be tested. The function returns a nonzero (true) value if it detects an error; otherwise, it returns zero (false). The function returns an integer value.

The **ferror** function tests for errors before performing a subsequent read or write to the file. For example, in the following program fragment, **ferror** tests the file given by **stream**:

```
#include <stdio.h>

main()
{
   char storage[5];
   FILE *stream;

   if ((stream = fopen("fwfile", "r")) == NULL)
   {
      fprintf(stderr, "cannot open fwfile\n");
      exit(1);
   }
   if (!ferror(stream))
      fread(storage, sizeof(storage), 1, stream);
}
```

If **ferror** returns zero, the next item in the file given by stream is copied to **x**. Otherwise, execution passes to the next statement.

Warning: Further use of a file after an error is detected may cause undesirable results.

Closing a File

The **fclose** function closes a file by breaking the connection between the file pointer and the structure created by **fopen**. Closing a file empties the contents of the corresponding buffer and frees the file pointer for use by another file. The function call has the form:

```
fclose(stream)
```

where *stream* is the file pointer of the file to be closed. The function normally returns 0, but returns EOF if it encounters an error.

The **fclose** function frees file pointers when they are no longer needed. This is important because usually no more than 20 files can be open at the same time. For example, the following program fragment closes the file given by **infile** when the file has reached its end:

```
FILE *infile;

if (feof(infile))
   fclose(infile);
```

Whenever a program terminates normally, PC XENIX calls the **fclose** function for each open file. You do not need to call **fclose** unless the program must close a file before its end. Also, the function calls **fflush** to ensure that everything written to the file's buffer gets to the file.

Program Example

This section shows how you may use the stream functions you have seen so far to perform useful tasks. The following program counts the characters, words, and lines found in one or more files, and uses the **fopen**, **fprintf**, **getc**, and **fclose** functions to open, write, read, and close the given files. The program incorporates a basic design that is common to other PC XENIX programs. It uses the file names found in the command line as the files to open and read or, if no names are present, it uses the standard input. This allows the program to start on its own or to be the receiving end of a pipe.

```
#include <stdio.h>

#define FALSE 0
#define TRUE  1

main(argc, argv)  /* wc: count lines, words, chars */
int argc;
char *argv[ ];
{
   int c, i = 1, inword;
   FILE *fp, *fopen();
   long tlines, lines, twords, words, tchars, chars;

   fp = stdin;
   tlines = 0, twords = 0, tchars = 0;
   do
   {
      if (argc > 1 && (fp = fopen(argv[i], "r")) == NULL)
      {
         fprintf(stderr, "%s:  can't open %s\n", argv[0], argv[i]);
         continue;
      }
      lines = words = chars = inword = 0;
      for (;;)
      {
         if ((c = getc(fp)) == EOF)
            if (feof(fp))
               break;
         chars++;
         if (c == '\n')
            lines++;
         if (c == ' ' || c == '\t' || c == '\n')
            inword = FALSE;
         else if (!inword)
         {
            inword = TRUE;
            words++;
         }
      }
      printf("%s: ", (argc > 1 ? argv[i] : "stdin"));
      printf("%d %d %d\n", lines, words, chars);
      fclose(fp);
      tlines += lines;
      twords += words;
      tchars += chars;
   } while (++i < argc);
   if (argc > 2)
      printf("Total (all files): %7ld %7ld %7ld\n",
              tlines, twords, tchars);
}
```

The program uses **fp** as the pointer to receive the current file pointer. Initially, we set **fp** to **stdin** in case no file names are present in the command line. If a file name is present, the program calls **fopen** and assigns the file pointer to **fp**. If **fopen** cannot open the file (in which case **fopen** returns **NULL**), the program writes an error message to the standard error file **stderr** with the **fprintf** function. The function prints the format string **wc: can't open %s**, replacing the %s with the name pointed to by **argv[*i*]**.

When **fopen** does open the file, the program uses the **getc** function to read each character from the file. As it reads characters, the program keeps a count of the number of characters, words, and lines. The program continues to read until it encounters the end of the file, that is, when **getc** returns the value **EOF**.

Once a file has reached its end, the program uses the **printf** function to display the character, word, and line counts at the standard output. The format string in this function causes **printf** to display the counts as decimal numbers with no more than seven digits. The program then closes the current file with the **fclose** function and examines the command line arguments to see if there is another file name.

When the program has counted all files, it uses the **printf** function to display a grand total at the standard output, then stops execution with the **exit** function.

Using More Stream Functions

The stream functions allow more control over a file than just opening, reading, writing, and closing. The functions also let a program take an existing file pointer and reassign it to another file (similar to redirecting the standard input and output files) as well as manipulate the buffer that is used for intermediate storage between the file and the program.

Using Buffered Input and Output

PC XENIX uses buffered input and output (I/O) to reduce the time needed to read from and write to files. Buffered I/O lets the system collect the characters to be read or written and then transfer them all at once rather than one character at a time. This reduces the number of times the system must access the I/O devices and consequently provides more time for running user programs. Not all files have buffers. For example, files associated with terminals, such as the standard input and output, do not have buffers; for these files, the system can transfer the input and output more quickly. When a file does have a buffer, the buffer size in bytes is given by the manifest constant **BUFSIZ**, which is defined in the **stdio.h** file.

When a file has a buffer, the stream functions read from and write to the buffer instead of the file. The system keeps track of the buffer and when necessary fills it with new characters (when reading) or flushes (copies) it to the file (when writing). Normally, a buffer is not directly accessible to a program; however a program can define its own buffer for a file with the **setbuf** function. (This is not recommended.) The function also lets a program change a buffered file to an unbuffered one. The **ungetc** function lets a program put a character it has read back into the buffer, and the **fflush** function lets a program flush the buffer before it is full.

Reopening a File

The **freopen** closes the file associated with a given file pointer, then opens a new file, and gives it the same file pointer as the old file. The function call has the form:

freopen(*newfile*, *type*, *stream*)

where *newfile* is a pointer to the name of the new file, *type* is a pointer to the string that defines how the file is to be opened (**r** for reading, **w** for writing, and **a** for appending), and *stream* is the file pointer of the old file. The function returns the file pointer *stream* if the function opens the new file. Otherwise, the function returns the null pointer value **NULL**.

Use the **freopen** function to attach the predefined file pointers **stdin**, **stdout**, and **stderr** to other files. For example, the following program fragment opens the file named by **newfile** as the new standard output file:

```
char *newfile;
FILE *nfile, *freopen();

nfile = freopen(newfile, "w", stdout);
```

This has the same effect as using the redirection symbols in the command line of the program.

Setting the Buffer

The **setbuf** function changes the buffer associated with a given file to the program's own buffer. It can also change the access to the file to no buffering. The function call has the form:

setbuf(*stream*, *buf*)

where *stream* is a file descriptor and *buf* is a pointer to the new buffer or is the null pointer value **NULL** if you do not want a buffer. If you give a buffer, it must be at least **BUFSIZ** bytes in length. You find **BUFSIZ** in **stdio.h**.

The function creates a buffer for the standard output when it is assigned to the user's terminal, which improves execution time by eliminating the need to write one character to the screen at a time. For example, the following program fragment changes the buffer of the standard output to the location pointed at by **p**:

```
char *p, *malloc();


if (p = malloc(BUFSIZ))
   setbuf(stdout, p);
```

The new buffer (`p`) is **BUFSIZ** bytes long.

You also can use the function to change a file from buffered to unbuffered input or output. Unbuffered input and output generally increase the total time needed to transfer large numbers of characters to or from a file.

Call the **setbuf** function immediately after opening a file and before reading or writing to it. Furthermore, use the **fclose** or **fflush** function to flush the buffer before terminating the program. If you do not use one of these functions, some data written to the buffer might not be written to the file.

Putting a Character Back into a Buffer

The **ungetc** function puts a character back into the buffer of a given file. The function call has the form:

`ungetc(`*c*`, `*stream*`)`

where *c* is the character to put back and *stream* is the file pointer of the file. The function normally returns the same character it put back, but returns the value **EOF** if it encounters an error.

The function scans a file for the first character of a string of characters. For example, the following program fragment puts the first character that is not a whitespace character back into the buffer of the file given by **infile**, allowing the subsequent call to **gets** to read that character as the first character in the string:

```
#include <stdio.h>
#include <ctype.h>

FILE *infile;
char name[20];
int c;

while (isspace(c = getc(infile)))
   ;

if (!feof(infile))
{
   ungetc(c, infile);
   fgets(name, 19, infile);
}
```

Putting a character back into the buffer does not change the corresponding file. It only changes the next character to be read.

The function can put a character back only if one has been previously read. The function cannot put more than one character back at a time. This means if 3 characters are read, **ungetc** can put only the last character back, never the first 2.

Never put the value **EOF** back in the buffer.

Note: The **seek** function may erase memory of the push-back character.

Flushing a File Buffer

The **fflush** function empties the buffer of a given file by immediately writing the buffer contents to the file. The function call has the form:

`fflush(`*stream*`)`

where *stream* is the file pointer of the file. The function normally returns zero, but returns the value **EOF** if it encounters an error.

The function guarantees that the contents of a full or partially filled buffer are written to the file. For example, the following program fragment empties the buffer for the file given by **outtty** if the error condition given by **errflag** is 0:

```
FILE *outtty;
int errflag;

if (errflag == 0)
   fflush(outtty);
```

The **fclose** function automatically calls the **fflush** function to empty the buffer before closing the file. This means that you do not need to explicitly call **fflush** if you are also closing the file.

The function ignores any attempt to empty the buffer of a file opened for reading.

Using the Low-Level Functions

The low-level functions provide direct access to files and peripheral devices. They are direct calls to the routines used in PC XENIX to read from and write to files and peripheral devices. The low-level functions give a program the same control over a file or device as the system, letting it access the file or device in ways that the stream functions do not. However, low-level functions, unlike stream functions, do not provide buffering or any other useful services of the stream functions. This means that any program that uses low-level functions has the complete burden of handling input and output.

Like the stream functions, you cannot use the low-level functions to read from or write to a file until you have opened the file. A program may use the **open** function to open an existing or a new file. You can open a file for reading, writing, or appending.

Once you open a file for reading, a program can read bytes from it with the **read** function. A program can write to a file opened for writing or appending with the **write** function. A program can close a file with the **close** function.

Note: For increased portability, use the **stdio** functions instead of the low-level functions.

Using File Descriptors

Each file that you have opened for access with the low-level functions has a unique integer called a file descriptor associated with it. A file descriptor is similar to a file pointer in that it identifies the file. A file descriptor, unlike a file pointer, does not point to any specific structure. Instead, the descriptor is used internally by the system to access the necessary information. Because the system maintains all information about a file, the only access to a file for a program is through the file descriptor.

There are three predefined file descriptors (just as there are three predefined file pointers) for the standard input, output, and error files. The descriptors are 0 for the standard input, 1 for the standard output, and 2 for the standard error file. As with predefined file pointers, a program can use the predefined file descriptors without explicitly opening the associated files.

If the standard input and output files are redirected, the system changes the default assignments for the file descriptors 0 and 1 to the named files. This is also true if the input or output is associated with a pipe. File descriptor 2 normally remains directed to the terminal.

Opening a File

The **open** function opens an existing or a new file and returns a file descriptor for that file. The function call has the form:

fd `=` `open(`*name*`,` *access*`[,` *mode*`])`

where *fd* is the integer variable to receive the file descriptor; *name* is a pointer to a string containing the file name; *access* is an integer expression giving the type of file access; and *mode* is an integer number giving a new file's permissions. The function normally returns a file descriptor (a positive integer) but returns -1 if it encounters an error.

Form the *access* expression by using one or more of the following manifest constants:

O_RDONLY	For reading
O_WRONLY	For writing
O_RDWR	For both reading and writing
O_APPEND	For appending to the end of an existing file
O_CREAT	For creating a new file.

You can combine the constants with the bitwise OR operator (|). You need *mode* only if you give **O_CREAT**. For example, in the following program fragment, the function opens the existing file named **/usr/accounts** for reading and opens the new file named **/usr/tmp/scratch** for reading and writing:

```
#include <fcntl.h>

int in, out;

in = open("/usr/accounts", O_RDONLY);
out = open("/usr/tmp/scratch", O_WRONLY | O_CREAT, 0754);
```

Each file has a set of permissions associated with it, that is, 9 bits of protection information that control who can read, write, and execute the file. These 9 bits consist of 3 bits that determine what permissions the owner of the file has, 3 bits that determine what permissions the owner's group has, and 3 bits that determine what permissions all others have. A 3-digit octal number is the most convenient way to specify the permissions. In the example above, the octal number 0754 specifies read, write, and execute permission for the owner, read and execute permission for the group, and read for everyone else.

Warning: If you give **O_CREAT** and the file already exists, the function destroys the old contents of the file.

Reading Bytes from a File

The **read** function reads 1 or more bytes of data from a given file and copies them to a given memory location. The function call has the form:

n_read = `read(`*fd*`,` *buf*`,` *n*`)`

where *n_read* is the variable to receive the count of bytes actually read; *fd* is the file descriptor of the file; *buf* is a pointer to the memory location to receive the bytes read; and *n* is a count of the desired number of bytes for the function to read. The function normally returns the same number of bytes as requested but returns fewer if the file does not have that many bytes left for the function to read. The function returns 0 if the file reaches its end or -1 if it encounters an error.

When the file is a terminal, **read** normally reads only up to the next newline.

The number of bytes for the function to read is arbitrary. The two most common values are 1, which means one character at a time, and 512, which corresponds to the physical block size on many peripheral devices.

Writing Bytes to a File

The **write** function writes 1 or more bytes from a given memory location to a given file. The function call has the form:

n_written `=` `write(`*fd*`,` *buf*`,` *n*`)`

where *n_written* is the variable to receive a count of bytes actually written; *fd* is the file descriptor of the file; *buf* is the name of the buffer containing the bytes for the function to write; and *n* is the number of bytes for the function to write.

The **write** function always returns the number of bytes written. It is an error if the return value is not equal to the number of bytes you requested the function to write.

The number of bytes for the function to write is arbitrary. The two most common values are 1, which means one character at a time and 512, which corresponds to the physical block size on many peripheral devices.

Closing a File

The **close** function breaks the connection between a file descriptor and an open file and frees the file descriptor for use with some other file. The function call has the form:

`close(`*fd*`)`

where *fd* is the file descriptor of the file to close. The function normally returns 0 but returns -1 if it encounters an error.

The **close** function closes files that you no longer need. For example, the following program fragment closes the standard input if the argument count is greater than 1:

```
int fd;

if (argc > 1)
   close(0);
```

All open files in a program close when a program terminates normally or when the **exit** function is called. In these cases, you do not need to explicitly call the **close** function.

Program Examples

This section shows how to use the low-level functions to perform useful tasks and presents three examples that incorporate the functions as the sole method of input and output.

The first program copies its standard input to its standard output:

```
#include <stdio.h>

main()  /* copy input to output */
{
   char buf[BUFSIZ];
   int n;

   while ((n = read( 0, buf, BUFSIZ)) > 0)
      write(1, buf, n);
}
```

The program uses the **read** function to read **BUFSIZE** bytes from the standard input (file descriptor 0). It then uses **write** to write the same number of bytes it read to the standard output (file descriptor 1). If the standard input file size is not a multiple of **BUFSIZE**, the last **read** returns a smaller number of bytes for the **write** function to write, and the next call to **read** returns zero.

You can use this program like a copy command to copy the content of one file to another. You can do this by redirecting the standard input and output files.

The second example shows how you can use the **read** and **write** functions to construct higher level functions like **getchar** and **putchar**.

For example, the following is a version of **getchar** that performs unbuffered input:

```
/* do NOT include <stdio.h> with this version */

#define EOF -1

getchar()         /* unbuffered single character input */
{
   char c;

   return((read(0, &c, 1) > 0) ? c : EOF);
}
```

You must declare the variable **c char** because **read** accepts a character pointer. In this case, mask the character you want returned with octal 0377 to ensure that it is positive; otherwise, sign extension may make it negative.

The second version of **getchar** reads input in large blocks but hands out the characters one at a time:

```
/* do NOT include <stdio.h> with this version */

#define BUFSIZE 512
#define EOF -1

getchar()               /* buffered version       */
{
   static char buf[BUFSIZE];
   static char *bufp = buf;
   static int n = 0;

   if (n == 0)          /* buffer is empty        */
   {
      n = read(0, buf, BUFSIZE);
      bufp = buf;
   }
   return((--n >= 0) ? *bufp++ : EOF);
}
```

Again, mask each character with the octal constant 0377.

The third example shows a simplified version of the utility **cp**, a program that copies one file to another.

The main simplification is that this version copies only one file and does not permit the second argument to be a directory:

```
#include <stdio.h>
#include <fcntl.h>

#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv)   /* cp: copy file 1 to file 2         */
int argc;
char *argv[ ];
{
   int f1, f2, n;
   static char buf[BUFSIZ];
   static char nullstring[1] = "";

   if (argc != 3)
      error("usage:  cp from to", nullstring);
   if ((f1 = open(argv[1], O_RDONLY)) == -1)
      error("can't open", argv[1]);
   if ((f2 = open(argv[2], O_CREAT ¦ O_WRONLY,
                  PMODE)) == -1)
      error("can't create", argv[2]);
   while ((n = read(f1, buf, BUFSIZ)) > 0)
      if (write(f2, buf, n) != n)
         error("write error on", argv[2]);
   exit(0);
}

error(s1, s2)      /* print error message and exit */
char *s1, *s2;
{
   printf("cp: %s %s\n",s1, s2);
   exit(1);
}
```

There is a limit (usually 20) to the number of files that a program may open simultaneously. Therefore, any program that intends to process many files must prepare to reuse file descriptors by closing unneeded files.

Using Random Access I/O

Input and output operations on any file are normally sequential. This means each read or write takes place at the character position immediately after the last character read or written. The standard library, however, provides a number of stream and low-level functions that allow a program to access a file randomly, that is, to exactly specify the position it next wishes to read from or write to.

The functions that provide random access operate on the character pointer of a file. Every open file has a character pointer that points to the next character to be read from that file, or the next place in the file to receive a character. Normally, the system maintains and controls the character pointer, but the random access functions let a program move the pointer to any position in the file.

Moving the Character Pointer

The **lseek** function, a low-level function, moves the character pointer in a file opened for low-level access to a given position. The function call has the form:

`lseek(`*fd*`,` *offset*`,` *origin*`)`

where *fd* is the file descriptor of the file, *offset* is the number of bytes to move the character pointer, and *origin* is the number that gives the starting point for the move. It may be 0 for the beginning of the file, 1 for the current position, and 2 for the end.

For example, this call forces the current position in the file whose descriptor is 3 to move to the 512th byte from the beginning of the file:

```
lseek(3, (long)512, 0);
```

Subsequent reading or writing begins at that position. The value of *offset* must be a long integer, and *fd* and *origin* must be integers.

The function can move the character pointer to the end of a file to allow appending, or it can reset the indicated file back to the beginning of the file as in a **rewind** function. For example, the call:

```
lseek(fd, (long)0, 2);
```

prepares the file for appending, and:

```
lseek(fd, (long)0, 0);
```

rewinds the file (moves the character pointer to the beginning). Notice the (**long**) 0 argument; you also could write it as:

```
0L
```

Using **lseek** allows you to treat files more or less like large arrays, but at the price of slower access. For example, the following function reads any number of bytes from any arbitrary place in a file:

```
get(fd, pos, buf, n)  /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
   if (lseek(fd, pos, 0) == -1)
      return(-1);

   return(read(fd, buf, n));
}
```

Moving the Character Pointer in a Stream

The **fseek** function, a stream function, moves the character pointer in a file to a given location. The function call has the form:

`fseek(`*stream*`,` *offset*`,` *ptrname*`)`

where *stream* is the file pointer of the file, *offset* is the number of characters to move to the new position (it must be a long integer), and *ptrname* is the starting position in the file of the move (0 for beginning, 1 for current position, or 2 for end of the file). The function normally returns zero but returns a nonzero value if it encounters an error.

For example, the following program fragment moves the character pointer to the end of the file given by **stream**:

```
FILE *stream;

fseek(stream, (long)0, 2);
```

You can use the function on either buffered or unbuffered files. **fseek** must be opened using **fopen**.

Note: **fseek** undoes the effects of **ungetc**.

Rewinding a File

The **rewind** function, a stream function, moves the character pointer to the beginning of a given file. The function call has the form:

`rewind(`*stream*`)`

where *stream* is the file pointer of the file. The function is equivalent to the following function call:

```
fseek(stream, 0L, 0);
```

It is a more readable version of the call.

Note: **fseek** undoes the effects of **ungetc**.

Getting the Current Character Position

The **ftell** function, a stream function, returns the current position of the character pointer in the given file. The returned position is always relative to the beginning of the file. The function call has the form:

p `= ftell(`*stream*`)`

where *stream* is the file pointer of the file and *p* is the variable to receive the position. The return value is always a long integer. The function returns the value -1 if it encounters an error.

The **ftell** function saves the current location in the file so that you can later return to that position.

Chapter 4. Screen Processing

This chapter explains how to use the screen updating and cursor movement library named **curses**. The library provides functions to:

- Create screen windows
- Update screen windows
- Receive input from the terminal in a screen-oriented way
- Optimize the motion of the cursor on the screen.

Terminology

In this chapter, we use the following terminology:

window	An internal representation containing an image of what a section of the terminal screen may look like at some point in time. This subsection can either encompass the entire terminal screen, or any smaller portion down to a single character within that screen.
terminal	Sometimes called terminal screen. The current appearance of the terminal's screen; in other words, what the user sees now.
screen	This is a subset of windows that are as large as the terminal screen; in other words, they start at the upper-left corner and encompass the lower-right corner. One of these, **stdscr**, is automatically provided for the programmer.

The screen processing library has a variety of predefined names. These names refer to variables, manifest constants, and types that you can use with the library functions. The names, types, and descriptions of the variables in the file **usr/include/curses.h** are:

Variables

Name	**Type**	**Description**
curscr	WINDOW *	A pointer to the current version of the terminal screen.
stdscr	WINDOW *	A pointer to the default screen used for updating when you do not define an explicit screen.
Def_term	char *	A pointer to the default terminal type if the type cannot be determined.
My_term	bool	The terminal type flag. If set, it causes the terminal specification in **Def_term** to be used, regardless of the real terminal type.
ttytype	char *	A pointer to the full name of the current terminal.
LINES	int	The number of lines on the terminal.
COLS	int	The number of columns on the terminal.
ERR	Boolean false value (0)	The error flag. Returned by functions on an error.
OK	Boolean true value (1)	The okay flag. Returned by functions on successful operation.

The names and descriptions of the types and constants are:

Types and Constants

Name	**Description**
bool	A type. It is the same as char type.
FALSE	The Boolean false value (0).
reg	A storage class. It is the same as register storage class.

TRUE	The Boolean true value (1).
WINDOW	A structure of short integers, characters, and pointers. This represents current, maximum, and beginning x and y positions, clear, leave, and scroll toggles, three pointers to first and last characters and the y pointer.

Screen Processing Overview

Screen processing gives a program a simple and efficient way to use the capabilities of the terminal attached to the program's standard input and output files. Screen processing does not rely on the terminal type. Instead, the screen processing functions use the terminal capability file **/etc/termcap** to tailor their actions for any given terminal. This can make a screen processing program more terminal-independent. You can run the program with any terminal as long as you describe that terminal in the **/etc/termcap** file.

The screen processing functions access a terminal screen by working through intermediate screens and windows in memory. A screen is a representation of what the entire terminal screen should look like. A window is a representation of what some portion of the terminal screen should look like. One or more windows can make up a screen. A window can be as small as a single character or as large as an entire screen.

Before you can use a screen or window, you must create it by using the **newwin** or **subwin** functions. These functions define the size of the screen or window in terms of lines and columns. Each position in a screen or window represents a place for a single character and corresponds to a similar place on the terminal screen. Positions are numbered according to line and column. For example, the position in the upper-left corner of a screen or window is (0,0) and the position immediately to its right is (0,1). A screen can have a maximum of 66 lines and 80 columns. Its upper-left corner corresponds to the upper-left corner of the terminal screen. A window, on the other hand, can be any size (within the limits of the screen). Its upper-left corner can correspond to any position on the terminal screen. For convenience, the **initscr** function, which initializes a program for screen processing, also creates a default screen, **stdscr** (for standard screen). You can use **stdscr** without first creating it.

The function also creates **curscr** (for current screen), which contains a copy of what is currently on the terminal screen.

To display characters at the terminal screen, a program must write these characters to a screen or window using screen processing functions such as **addch** and **waddch**. If necessary, a program can move to the desired position in the screen or window by using the **move** and **wmove** functions. Once characters are added to a screen or window, the program can copy the characters to the terminal screen by using the **refresh** or **wrefresh** function. These functions update the terminal screen according to what has changed in the given screen or window. Because the terminal screen is not changed until a program calls **refresh** or **wrefresh**, a program can maintain several windows, each containing different characters for the same portion of the terminal screen. The program can choose which window should actually be displayed before updating the screen.

A program can continue to add new characters to a screen or window as needed and edit these characters by using functions such as **insertln**, **deleteln**, and **clear**. A program can also combine windows to make a composite screen using the **overlay** and **overwrite** functions. In each case, the **refresh** or **wrefresh** function copies the changes to the terminal screen.

Using the Library

To use the **curses** library in a program, you must add the line:

```
#include <curses.h>
```

to the beginning of your program. The **curses.h** file contains definitions for types and variables used by the library.

The screen processing functions are in the library files **libcurses.a** and **libtermcap.a**. These files are not automatically read when you compile your program, so you must include the appropriate library switches in your invocation of the compiler. The command line must have the form:

```
cc file . . . -lcurses -ltermcap
```

where *file* is the name of the source file you wish to compile. You may give more than one file name if desired. You may also use other compiler options in the command line. For example, the command:

```
cc main.c intf.c -lcurses -ltermcap -o sample
```

compiles the files **main.c** and **intf.c**, and copies the executable program to the file **sample** after linking the screen processing library files to the program.

The **curses.h** file includes the file **sgtty.h** in your program. Do not include this file twice.

Screen Updating

To update the screen optimally, the routines must know what the screen currently looks like and what the programmer wants it to look like next. For this purpose, a data type (structure) named **WINDOW** is defined that describes a window image to the routines. This includes its starting position on the screen, the (y,x) coordinates of the upper-left corner, and its size. One of these (called **curscr** for current screen) is a screen image of what the terminal currently looks like. Another screen (called **stdscr** for standard screen) is provided by default to make changes on.

A window is a purely internal representation. Use it to build and store a potential image of a portion of the terminal screen. The window does not necessarily have any relation to what is really on the terminal screen. It is more like an array of characters on which to make changes.

If a window describes what some part of the screen should look like and it is not part of the **stdscr**, then call the **refresh()** or **wrefresh()** routine. The **refresh()** routine makes the terminal, in the area covered by the window, look like that window. Changing something on a window does not change the terminal screen. You can make actual updates to the terminal screen only by calling **refresh()** or **wrefresh()**. This allows you to maintain several ideas of what a portion of the terminal screen should look like. Also, you can make changes to windows in any order, without regard to motion efficiency. You can effectively say "make it look like this," and let the routine worry about the best way to do this.

Naming Conventions

The routines can use several windows, but two are given: **curscr**, which knows what the terminal looks like, and **stdscr**, which is what you want the terminal to look like next. The screen package also uses the standard I/O library, so <**curses.h**> includes <**stdio.h**>. It is redundant, but harmless, for you to also include <**stdio.h**>. You should never access **curscr** directly. Make changes to the appropriate screen, and then call the routine **refresh()** or **wrefresh()**.

Many functions deal with **stdscr** as a default screen. For example, to add a character to **stdscr**, call **addch()** with the desired character. If you want to use a different window, use the routine **waddch()** for window-specific **addch()**. Actually, **addch()** is a **#define** macro with arguments, as are most of the "functions" that deal with **stdscr** as a default. This convention of prefixing function names with a "w" when you want to use them with specific windows is consistent. The only routines that do not do this are those with which you must always specify a window.

To move the current (y,x) coordinates from one point to another, use the routines **move()** and **wmove()**. However, it is often desirable to first move and then perform some I/O operation. To avoid clumsiness, you can precede most I/O routines with the prefix "mv" and then add the desired (y,x) coordinates to the arguments to the function. For example, you can replace the calls:

```
move(y,x);
addch(ch);
```

by:

```
mvaddch(y,x,ch);
```

and:

```
wmove(win, y, x);
waddch(win, ch);
```

by:

```
mvwaddch(win, y, x, ch);
```

The window description pointer (*win*) comes before the added (*y*, *x*) coordinates. If you need such pointers, they are always the first parameters passed.

Preparing the Screen

The **initscr** and **endwin** functions perform the operations required to initialize and terminate programs that use the screen processing functions. The following sections describe these functions and how they affect the terminal.

Initializing the Screen

The **initscr** function initializes screen processing for a program by allocating the required memory space for the screen processing functions and variables and by setting the terminal to the proper modes. The function call has the form:

```
initscr()
```

This function call does not require arguments.

When to Use initscr()

To use the screen package, the routines must know about terminal characteristics, and the space for **curscr** and **stdscr** must be allocated. The **initscr()** function performs these functions. Because it must allocate space for the windows, it can overflow memory when attempting to do so. On this rare occasion, **initscr()** returns **ERR**. You must always call the **initscr()** function before you use any of the routines that affect windows. If **initscr()** is not called first, the program dumps memory as soon as either **curscr** or **stdscr** are referenced. However, it is usually best to wait to call **initscr()** until after you are sure you need it (for example, after checking for startup errors). Do not call terminal status changing routines like **nl()** and **crmode()** until after calling **initscr()**.

You must use the **initscr** function to prepare the program for subsequent calls to other screen processing functions and for use of the screen processing variables. For example, in the following program fragment, **initscr** initializes the screening processing functions:

```
#include <curses.h>
main()
{
initscr();
if (strcmp(ttytype, "dumb") == 0)
   fprintf(stderr, "Terminal type can't display screen.\n");
```

In this example, the predefined variable **ttytype** is checked for the current terminal type.

Using Terminal Capability and Type

The **initscr** function uses the terminal capability descriptions given in the **/etc/termcap** file to prepare the screen processing functions for creating and updating terminal screens. The descriptions define the character sequences required to perform a given operation on a given terminal. The screen processing functions use these sequences to add, insert, delete, and move characters on the screen. The descriptions are automatically read from the file when screen processing starts, so you do not need to directly access them with a program. For example, HO is a string that moves the cursor to the "home" position. These names are identical to those variables used in the **/etc/termcap** database to describe each capability. As there are two types of variables involving ttys, there are two routines. The first, **gettmode()**, sets some variables based upon the tty modes accessed by **gtty(2)**. The second, **setterm()**, accomplishes a larger task by reading in the descriptions from the **/etc/termcap** database. This example shows how **initscr()** uses these routines:

```
if (isatty(0))
{
   gettmode();
   if (sp = getenv("TERM"))
      setterm(sp);
}
else
   setterm(Def_term);
_puts(TI);
_puts(VS);
```

The **isatty()** function checks to see if file descriptor 0 is a terminal. The **isatty()** function is defined in the default C library function routines. It does a **gtty()** on the descriptor and checks the return value. If the return value is true, **gettmode()** sets the terminal description modes from a **gtty()**; **getenv()** gets the name of the terminal, and that value (if there is one) is passed to **setterm()**, which reads in the variables from **/etc/termcap** associated with that terminal. (The **getenv()** function returns a pointer to a string containing the name of the terminal, which is saved in the character pointer **sp**.) If **isatty()** returns false, the default terminal **Def_term** is used. The TI and VS sequences initialize the terminal. The **_puts()** routine is a macro that uses **tputs()**.

The **initscr** function uses the shell's TERM variable to determine which terminal capability description to use. The TERM variable is usually assigned an identifier when a user logs in. This identifier defines the terminal type and is associated with a terminal capability description in the **/etc/termcap** file.

If the TERM variable has no value, the functions use the default terminal type in the library's predefined variable **Def_term**. This variable initially has the value "dumb" (for dumb terminal), but the user may change it to any desired value. You must do this before calling the **initscr** function.

In some cases, it is desirable to force the screen processing functions to use the default terminal type. You can do this by setting the library's predefined variable **My_term** to the value 1. The predefined variable **ttytype** stores the full name of the current terminal.

Capabilities from termcap

Capabilities from **termcap** are of three kinds: string-valued options, numeric-valued options, and Boolean options. The string-valued options are the most complicated, because they may include padding information, which we describe now.

Intelligent terminals often require padding on intelligent operations at high (and sometimes even low) speed. A number before the string in the capability specifies those terminals that require padding. This normally is a number of milliseconds to pad the operation. In the current system, which has no true programmable delays, this is done by sending a sequence of pad characters (normally nulls).

In some cases, the pad is better computed as some number of milliseconds times the number of affected lines (to the bottom of the screen usually, except when terminals have insert modes that shift several lines). This is specified as, for example, 12* before the capability, to indicate 12 milliseconds per affected line. Capabilities where this makes sense are marked with P*.

Terminal capabilities, types, and identifiers are described in detail under **termcap** in *XENIX System Reference.*

Using Default Terminal Modes

The **initscr** function automatically sets a terminal to default operation modes. These modes define how the terminal displays characters sent to the screen and how it responds to characters typed at the keyboard. The **initscr** function sets the terminal to **ECHO** mode, which causes characters typed at the keyboard to appear on the screen, and **RAW** mode, which causes characters to be used as direct input (no editing or signal processing is done).

You can change the default terminal modes by using the appropriate functions described in "Setting a Terminal Mode" on page 4-39. If you change the modes, you must change them immediately after calling **initscr**. We describe terminal modes in detail under **tty** in *XENIX System Reference.*

Note: Use terminal mode functions only with other screen processing functions. Never use them alone.

Using Default Window Flags

The **initscr** function automatically clears the cursor, scroll, and clear flags of the standard screen to their default values. These flags, called the window flags, define how the **refresh** function affects the terminal screen when refreshing from the standard screen. When clear, the cursor flag prevents the terminal's cursor from moving back to its original location after the screen is updated, the scroll flag prevents scrolling on the screen, and the clear flag prevents the characters on the screen from being cleared before being updated. You can change the flags by using the functions described in "Setting Window Flags" on page 4-37.

Using the Default Terminal Size

The **initscr** function sets the terminal screen size to a default number of lines and columns. The predefined variables LINES and COLS gives the default values. You can change the default size of a terminal by setting the variables to new values. You should do this before the first call to **initscr**. If you do it after the first call, you must make a second call to **initscr** to delete the existing standard screen and create a new one.

Terminating Screen Processing

To do certain optimizations, and on some terminals to work at all, certain tasks must be done before the screen routines start up. The **gettmode()** and **setterm()** functions, which **initscr()** calls, perform these tasks. To clean up after the routines, use the **endwin()** function. It restores tty modes to what they were when you first called **initscr()**. Thus, anytime after the call to **initscr**, you should call **endwin()** before exiting.

The **endwin** function terminates the screen processing in a program by freeing all memory resources allocated by the screen processing functions and restoring the terminal to its state before screen processing began. The function call has the form:

```
endwin()
```

This function call does not require arguments.

You must use the **endwin** function before leaving a program that has called the **initscr** function to restore the terminal to its previous state. The function is generally the last function call in the program. For example, in the following program fragment **initscr** and **endwin** form the beginning and end of the program:

```
#include <curses.h>

main()
{
   initscr();
/* Program body. */
   endwin();
}
```

You must not call the **endwin** function if you have not called the **initscr** function. However, you must call the **endwin** function if the **gettmode** and **setterm** functions have been called, even if you have not called the **initscr** function. Also, you should not call the **endwin** function before any call to the **exit** function.

How to Use the Screen Package

This section describes how to apply the screen package to **stdscr**. All instructions work on any window by changing the function name and parameters, as mentioned earlier in “Naming Conventions.”

Once **initscr()** has allocated the screen windows, you can set them up for the run. If you want to allow the window to scroll, use **scrollok()**. If you want to leave the cursor after the last change, use **leaveok()**. If this isn't done, **refresh()** moves the cursor to the window's current (y,x) coordinates after updating it. You can create new windows of your own, too, by using the functions **newwin()** and **subwin()**. The **delwin()** function allows you to get rid of old windows. If you wish to change the official size of the terminal by hand, just set the variables LINES and COLS to be what you want, and then call **initscr()**. You should set these variables before the first call to **initscr()**, because the change deletes any existing **stdscr** and/or **curscr** before creating new ones.

Output

After you initialize the screen, you can update text on the terminal. The basic functions used to change data in a window are **addch()** and **move()**. The **addch()** function adds a character at the current (y,x) coordinates, returning **ERR** if it would cause the window to scroll illegally. (Printing a character in the lower-right corner of a terminal that scrolls automatically is an error when scrolling is not allowed.) The **move()** function changes the current(y,x) coordinates to whatever you want them to be. It returns **ERR** if you try to move off the window when scrolling is not allowed. You can combine the two into **mvaddch()** to do both operations at once.

The other output functions, such as **addstr()** and **printw()**, all call **addch()** to add characters to the window.

After you have put the desired characters on the window and you are ready to make the terminal screen reflect the window, call **refresh()**.

To find changes quickly, **refresh()** assumes that any part of the window not changed since the last **refresh()** of that window has not been changed on the terminal, that is, that you have not refreshed a portion of the terminal with an overlapping window. If this is not the case, use the **touchwin()** function to make it appear that the entire window has been changed, thus making **refresh()** check the whole subsection of the terminal for changes.

If you call **wrefresh()** with **curscr**, it makes the screen look like **curscr** thinks it looks. This is useful for implementing a command that would redraw the screen.

Input

Input is essentially a mirror image of output. The complementary function to **addch()** is **getch()**, which, if you have set **ECHO**, calls **addch()** to echo the character. Because the screen package needs to know what is on the terminal at all times, if characters are to be echoed, the tty must be in **RAW** or **CBREAK** mode. If it is not, **getch()** sets it to be **cbreak**, and then reads in the character.

Using the Standard Screen

The following sections explain how to use the standard screen to display and edit characters on the terminal screen.

Adding a Character

The **addch** function adds a given character to the standard screen and moves the character pointer one position to the right. The function call has the form:

`addch(`*ch*`)`

where *ch* gives the character to be added and must have **char** type. For example, if the current position is (0,0), the function call:

```
addch('A');
```

places the letter A at this position and moves the pointer to (0,1).

If you give a new-line (`'\n'`) character, the function deletes all characters from the current position to the end of the line and moves the pointer one line down. If the new-line flag is set, the function deletes the characters and moves the pointer to the beginning of the next line. If you give a return ('\r'), the function moves the pointer to the beginning of the current line. If you give a tab ('\t'), the function moves the pointer to the next tab stop, adding enough spaces to fill the gap between the current position and the stop. Tab stops are placed at every 8 character positions.

The **addch** function returns **ERR** if it encounters an error, such as illegal scrolling.

Adding a String

The **addstr** function adds a string of characters to the standard screen, placing the first character of the string at the current position and moving the pointer one position to the right for each character in the string. The function call has the form:

addstr(*str*)

where *str* is a character pointer to the given string. For example, if the current position is (0,0), the function call:

```
addstr("line");
```

places the beginning of the string "line" at this position and moves the pointer to (0,4).

If the string contains new-line, return, or tab characters, the function performs the same actions as described for the **addch** function. If the string does not fit on the current line, the string is truncated.

The **addstr** function returns **ERR** if it encounters an error, such as illegal scrolling.

Printing Strings, Characters, and Numbers

The **printw** function prints one or more values on the standard screen, where a value may be a string, a character, or a decimal, octal, or hexadecimal number. The function call has the form:

printw(*fmt* [, *arg*] . . .)

where *fmt* is a pointer to a string that defines the format of the values, and *arg* is a value to be printed. If you give more than one *arg*, separate each argument from the preceding argument with a comma (,). For each *arg* given, there must be a corresponding format given in *fmt*. A format may be %s for string, %c for character, and %d, %o, or %x for a decimal, octal, or hexadecimal number, respectively. (Other formats are described under **printf** in *XENIX System Reference*.) If you give %s, the corresponding *arg* must be a character pointer. For other formats, you can give the actual value or a variable containing the value.

The function copies both numbers and strings to the standard screen at the same time. For example, if the current position is (0,0), the function call:

```
printw("%s %d", name, 15);
```

prints the name given by the variable **name** starting at position (0,0). It then prints the number 15 immediately after the name.

The **printw** function returns **ERR** if it encounters an error, such as illegal scrolling.

Reading a Character from the Keyboard

The **getch** function reads a single character from the terminal keyboard and returns the character as a value. The function call has the form:

```
c = getch()
```

where *c* is the variable to receive the character.

The function reads a series of individual characters. For example, in the following program fragment, **getch** reads and stores characters until it encounters a new line or the end of the file, or until the buffer size has been reached:

```
char c, p[MAX];
int i = 1;
initscr();

while ((c = getch()) != '\n' && c != EOF && c != ERR && i < MAX
   p[i++] = c;
endwin();
```

If you have set the terminal to **ECHO** mode, **getch** copies the character to the standard screen; otherwise, the screen remains unchanged. If you have not set the terminal to **RAW** or **NOECHO** mode, **getch** sets the terminal to **CBREAK** mode, then restores the previous mode after reading the character. We describe terminal modes later in the chapter.

The **getch** function returns **ERR** if it encounters an error, such as illegal scrolling.

Reading a String from the Keyboard

The **getstr** function reads a string of characters from the terminal keyboard and copies the string to a given location. The function call has the form:

`getstr(`*str*`)`

where *str* is a character pointer to the variable or location to receive the string. When typed at the keyboard, the string must end with a new-line character or with the end-of-file character. A null character replaces the extra character when the string is stored. It is the programmer's responsibility to ensure that *str* has adequate space to store the typed string.

The function reads names and other text from the keyboard. For example, in the following program fragment **getstr** reads a file name from the keyboard and stores it in the array **name**:

```
char name[20];

getstr(name);
```

If you have set the terminal to **ECHO** mode, **getstr** copies the string to the standard screen. If you have not set the terminal to **RAW** or **NOECHO** mode, the function automatically sets the terminal to **CBREAK** mode, then restores the previous mode after reading the character. (See "Terminal Modes" on page 4-39 for more information.)

The **getstr** function returns **ERR** if it encounters an error, such as illegal scrolling.

Reading Strings, Characters, and Numbers

The **scanw** function reads one or more values from the terminal keyboard and copies the values to given locations. A value may be a string, character, or decimal, octal, or hexadecimal number. The function call has the form:

`scanw(`*fmt*`, `*argptr1*`, `*argptr2*` . . . )`

where *fmt* is a pointer to a string defining the format of the values to be read, and *argptr* is a pointer to the variable to receive a value. If you give more than one *argptr*, separate each from the preceding item with a comma. For each *argptr* given, you must give a corresponding format in *fmt*. A format may be %s for string, %c for character, and %d, %o, or %x for a decimal, octal, or hexadecimal number, respectively. (Other formats are described under **scanf** in *XENIX System Reference*.)

The function reads a combination of strings and numbers from the keyboard. For example, in the following program fragment, **scanw** reads a name and a number from the keyboard:

```
char name[20];
int id;

scanw("%s %d", name, &id);
```

In this example, the character array **name** and the integer variable **id** store the input values.

If you have set the terminal to **ECHO** mode, the function copies the string to the standard screen. If you have not set the terminal to **RAW** or **NOECHO** mode, the function automatically sets the terminal to **CBREAK** mode, then restores the previous mode after reading the character.

The **scanw** function returns **ERR** if it encounters an error, such as illegal scrolling.

Moving the Current Position

The **move** function moves the pointer to the given position. The function call has the form:

`move(`*y*`, `*x*`)`

where *y* is an integer value giving the new row position, and *x* is an integer value giving the new column position. For example, if the current position is (0,0), the function call:

```
move(5,4);
```

moves the pointer to line 5, column 4.

The **move** function returns **ERR** if it encounters an error, such as illegal scrolling.

Inserting a Character

The **insch** function inserts a character at the current position and shifts the existing character (and all characters to its right) one position to the right. The function call has the form:

```
insch(c)
```

where *c* is the character you want to insert.

The function inserts a series of characters into an existing line. For example, in the following program, **insch** inserts the number of characters given by **cnt** into the standard screen at position (15,15):

```
#include <curses.h>

main()
{
   static char string[] = "series of characters";
   int count = sizeof(string) - 1;

   initscr();
   move(15,15);
   while (count >= 0)
   {
      if (insch(string[count]) == ERR)
      {
         fprintf(stderr, "error return from insch()\n");
         endwin();
         exit(1);
      }
      else
         --count;
   }
   refresh();
   endwin();
}
```

The **insch** function returns **ERR** if it encounters an error, such as illegal scrolling. For information about the **refresh** function, see “Refreshing from the Standard Screen” on page 4-21.

Inserting a Line

The **insertln** function inserts a blank line at the current position and moves the existing line (and all lines below it) down one line, causing the last line to move off the bottom of the screen. The function call has the form:

```
insertln()
```

This function call does not require arguments.

Use the function to insert additional lines of text in the standard screen. For example, in the following program fragment, **insertln** inserts a blank line when the count in **cnt** is equal to 79:

```
int cnt;

if (cnt == 79)
   insertln();
```

The function returns **ERR** if it encounters an error, such as illegal scrolling.

Deleting a Character

The **delch** function deletes the character at the current position and shifts all the characters to the right of the deleted character one position to the left. A space replaces the last character on the line. The function call has the form:

```
delch()
```

This function call does not require arguments.

The function deletes a series of characters from the standard screen. For example, in the following program fragment, **delch** deletes the character at the current position as long as the count in **cnt** is not 0:

```
int cnt;

while (cnt != 0)
{
   delch();
   cnt--;
}
```

Deleting a Line

The **deleteln** function deletes the current line and shifts the line below the deleted line (and all lines below it) one line up, leaving the last line on the screen blank. The function call has the form:

```
deleteln()
```

This function call does not require arguments.

Use the **deleteln** function to delete existing lines from the standard screen. For example, in the following program fragment, **deleteln** deletes a line from the standard screen if the count in **cnt** is 79:

```
int cnt;

if (cnt == 79)
   deleteln();
```

Clearing the Screen

The **clear** and **erase** functions clear all characters from the standard screen by replacing them with spaces. The functions prepare the screen for new text. The function call:

```
clear()
```

clears all characters from the standard screen, moves the pointer to (0,0), and sets the standard screen's clear flag. The flag causes the next call to the **refresh** function to clear all characters from the terminal screen. The function call:

```
erase()
```

clears the standard screen, but does not set the clear flag. For example, in the following program fragment, **clear** clears the screen if the input value is 12:

```
int c;

if ((c = getch()) == 12)
   clear();
```

Clearing a Part of the Screen

The **clrtobot** and **clrtoeol** functions clear one or more characters from the standard screen by replacing the characters with spaces. The functions prepare a part of the standard screen for new characters.

The **clrtobot** function clears the screen from the current position to the bottom of the screen. For example, if the current position is (10,0), to clear all characters from line 10 and all lines below line 10, use the function call:

```
clrtobot();
```

The **clrtoeol** function clears the standard screen from the current position to the end of the current line. For example, if the current position is (10,10), the function call:

```
clrtoeol();
```

clears all characters from (10,10) to (10,79). The characters at the beginning of the line remain unchanged.

Neither the **clrtobot** or **clrtoeol** functions change the current position of the character pointer.

Refreshing from the Standard Screen

The **refresh** function updates the terminal screen by copying one or more characters from the standard screen to the terminal. The function effectively changes the terminal screen to reflect the new contents of the standard screen. The function call has the form:

```
refresh()
```

This function call does not require arguments.

Use the **refresh** function solely to display changes to the standard screen. The function copies only those characters that have changed since the last call to **refresh** and leaves any existing text on the terminal screen. For example, the following program fragment calls **refresh** twice:

```
addstr("The first time.\n");
refresh();
addstr("The second time.\n");
refresh();
```

In this example, the first call to **refresh** copies the string *The first time.* to the terminal screen. The second call copies only the string *The second time.* to the terminal, because the original string has not been changed.

The **refresh** function returns **ERR** if it encounters an error, such as illegal scrolling. If it does encounter an error, the function attempts to update as much of the screen as possible without scrolling.

Creating and Using Windows

The following sections explain how to create and use windows to display and edit text on the terminal screen.

Creating a Window

The **newwin** function creates a window and returns a pointer that you can use in subsequent screen processing functions. The function call has the form:

win = `newwin(`*lines*, *cols*, *begin_y*, *begin_x*`)`

where *win* is the pointer variable to receive the return value. *lines* and *cols* are integer values that give the total number of lines and columns, respectively, in the window. *begin_y* and *begin_x* are integer values that give the line and column positions, respectively, of the upper left corner of the window when it appears on the terminal screen. The *win* variable must have type **WINDOW ***.

Use the function in programs that maintain a set of windows, displaying different windows at different times or alternating between windows as needed. For example, in the following program fragment, **newwin** creates a new window and assigns the pointer to this window to the variable **midscreen**:

```
WINDOW *midscreen;


midscreen = newwin(5, 10, 9, 35);
```

The window has 5 lines and 10 columns. The upper left corner of the window is position (9,35) on the terminal screen.

If either *lines* or *cols* is zero, the function automatically creates a window that has "LINES - *begin_y*" lines or "COLS - *begin_x*" columns, where LINES and COLS are the predefined constants giving the total number of lines and columns on the terminal screen. For example, the function call:

```
newwin(0, 0, 0, 0);
```

creates a new window whose upper left corner is at position (0,0) and that has LINES lines and COLS columns.

Note: You must not create windows that exceed the dimensions of the screen.

The **newwin** function returns the value **(WINDOW *) ERR** on an error, such as insufficient memory for the new window.

Creating a Subwindow

The **subwin** function creates a subwindow and returns a pointer to the new window. A subwindow is a window that shares all or part of the character space of another window and provides an alternate way to access the characters in that space. The function call has the form:

swin `=` `subwin(`*win*`,` *lines*`,` *cols*`,` *begin_y*`,` *begin_x*`)`

where *swin* is the pointer variable to receive the return value. *win* is the pointer to the window to contain the new subwindow. *lines* and *cols* are integer values that give the total number of lines and columns, respectively, in the subwindow. *begin_y* and *begin_x* are integer values that give the line and column position, respectively, of the upper left corner of the subwindow when it appears on the terminal screen. The *swin* variable must have type **WINDOW ***.

The function divides a large window into separate regions. For example, in the following program fragment, **subwin** creates the subwindow named **cmdmenu** in the lower part of the standard screen:

```
WINDOW *cmdmenu;

cmdmenu = subwin(stdscr, 5, 80, 19, 0);
```

In this example, changes to **cmdmenu** affect the standard screen as well.

The **subwin** function returns the value **(WINDOW *) ERR** on an error, such as insufficient memory for the new window.

Adding and Printing to a Window

The **waddch**, **waddstr**, and **wprintw** functions add and print characters, strings, and numbers to a given window.

The **waddch** function adds a given character to the given window and moves the character pointer one position to the right. The function call has the form:

waddch(*win*, *ch*)

where *win* is a pointer to the window to receive the character, and *ch* gives the character you want to add; *ch* must have char type. For example, if the current position in the window **midscreen** is (0,0), the function call:

```
waddch(midscreen, 'A');
```

places the letter A at this position and moves the pointer to (0,1).

The **waddstr** function adds a string of characters to the given window, placing the first character of the string at the current position and moving the pointer one position to the right for each character in the string. The function call has the form:

waddstr(*win*, *str*)

where *win* is a pointer to the window to receive the string, and *str* is a character pointer to the given string. For example, if the current position is (0,0), the function call:

```
waddstr(midscreen, "line");
```

places the beginning of the string “line” at this position and moves the pointer to (0,4).

The **wprintw** function prints one or more values on the given window, where a value may be a string, a character, or a decimal, octal, or hexadecimal number. The function call has the form:

wprintw(*win*, *fmt* [,*arg*] . . .)

where *win* is a pointer to the window to receive the values, *fmt* is a pointer to a string that defines the format of the values, and *arg* is a value you want to print. If you give more than one *arg*, you must separate each from the preceding with a comma. For each *arg* given, there must be a corresponding format given in *fmt*. A format may be %s for string, %c for character, and %d, %o, or %x for a decimal, octal, or hexadecimal number, respectively. (Other formats are described under **printf** in *XENIX System Reference*.) If you give %s, the corresponding *arg* must be a character pointer. For other formats, you can give the actual value or a variable containing the value.

The function copies both numbers and strings to the standard screen at the same time. For example, in the following program fragment, **wprintw** prints a name and then the number 15 at the current position in the window **midscreen**:

```
char *name;

wprintw(midscreen, "%s %d", name, 15);
```

When you give a new-line, return, or tab character to a **waddch**, **waddstr**, or **wprintw** function, the functions perform the same actions as described for the **addch** function. The functions return **ERR** if they encounter errors, such as illegal scrolling.

Reading and Scanning for Input

The **wgetch**, **wgetstr**, and **wscanw** functions read characters, strings, and numbers from the standard input file and usually echo the values by copying them to the given window.

The **wgetch** function reads a single character from the standard input file and returns the character as a value. The function call has the form:

c = `wgetch(`*win*`)`

where *win* is a pointer to a window, and *c* is the character variable to receive the character.

The function reads a series of characters from the keyboard. For example, in the following program, **wgetch** reads characters until a colon (:) is found:

```
#include <curses.h>

#define MAX 80

main()
{
   int i = 0;
   char c, input[MAX];
   WINDOW *cmdmenu;

   initscr();
   if ((cmdmenu = subwin(stdscr,5,80,10,0)) != (WINDOW *) ERR)
      while ((c = wgetch(cmdmenu)) != (char) ERR &&
              c != ':' && i < MAX)
          input[i++] = c;

   endwin();
}
```

The **wgetstr** function reads a string of characters from the terminal keyboard and copies the string to a given location. The function call has the form:

wgetstr(*win*, *str*)

where *win* is a pointer to a window, and *str* is a character pointer to the variable or location to receive the string. When typed at the keyboard, the string must end with a new-line character or with the end-of-file character. A null character replaces the extra character when the string is stored. It is the programmer's responsibility to ensure that *str* has adequate space for storing the typed string.

The function reads names and other text from the keyboard. For example, in the following program fragment, **wgetstr** reads a string from the keyboard and stores it in the array **filename**:

```
char filename[20];

wgetstr(cmdmenu, filename);
```

The **wscanw** function reads one or more values from the standard input file and copies the values to given locations. A value may be a string, a character, or a decimal, octal, or hexadecimal number. The function call has the form:

wscanw(*win*, *fmt* [,*argptr*] . . .)

where *win* is a pointer to a window, *fmt* is a pointer to a string defining the format of the values to be read, and *argptr* is a pointer to the variable to receive a value. If you give more than one *argptr*, you must separate each from the preceding by a comma. For each *argptr* given, there must be a corresponding format given in *fmt*. (The * format is an exception to this rule). A format may be %s for string, %c for character, and %d, %o, or %x for a decimal, octal, or hexadecimal number, respectively. (Other formats, including the * format, are described under **scanf** in *XENIX System Reference*.)

The function reads a combination of strings and numbers from the keyboard. For example, in the following program fragment, **wscanw** reads a name and a number from the keyboard:

```
char name[20];
int id;

wscanw(midscreen, "%s %d", name, &id);
```

In this example, the name is stored in the character array **name** and the number in the integer variable **id**.

If you have set the terminal to **ECHO** mode, the function copies the string to the given window. If you have not set the terminal to **RAW** or **NOECHO** mode, the function automatically sets the terminal to **CBREAK** mode, then restores the previous mode after reading the character.

These functions return **ERR** if they encounter errors, such as illegal scrolling.

Moving the Current Position in a Window

The **wmove** function moves the current position in a given window. The function call has the form:

wmove(*win*, *y*, *x*)

where *win* is a pointer to a window, *y* is an integer value giving the new line position, and *x* is an integer value giving the new column position. For example, the function call:

```
wmove(midscreen, 4, 4);
```

moves the current position in the window **midscreen** to (4,4).

The function returns **ERR** if it encounters an error, such as illegal scrolling.

Inserting Characters and Lines

The **winsch** and **winsertln** functions insert characters and lines into a given window.

The **winsch** function inserts a character at the current position and shifts the existing character (and all characters to its right) one position to the right. The function call has the form:

winsch(*win*, *c*)

where *win* is a pointer to a window, and *c* is the character you want to insert.

The function edits the contents of the given window. For example, the function call:

```
winsch(midscreen, 'X');
```

inserts the character **X** at the current position in the window **midscreen**.

The **winsertln** function inserts a blank line at the current position and moves the existing line (and all lines below it) down one line, causing the last line to move off the bottom of the window. The function call has the form:

winsertln(*win*)

where *win* is a pointer to the window to receive the blank line.

The function inserts lines into a window. For example, in the following program fragment, **winsertln** inserts a blank line at the top of the window **cmdmenu** preparing it for a new line:

```
char line[80];

wmove(cmdmenu, 3, 0);
winsertln(cmdmenu);
waddstr(cmdmenu, line);
```

Both functions return **ERR** if they encounter errors, such as illegal scrolling.

Deleting Characters and Lines

The **wdelch** and **wdeleteln** functions delete characters and lines from the given window.

The **wdelch** function deletes the character at the current position and shifts the character to the right of the deleted character (and all characters to its right) one position to the left. A space replaces the last character on the line. The function call has the form:

wdelch(*win*)

where *win* is a pointer to a window.

The function edits the contents of the standard screen. For example, the function call:

```
wdelch(midscreen);
```

deletes the character at the current position in the window **midscreen**.

The **wdeleteln** function deletes the current line and shifts the line below the deleted line (and all lines below it) one line up, leaving the last line on the window blank. The function call has the form:

wdeleteln(*win*)

where *win* is a pointer to a window.

The function deletes existing lines from a given window. For example, in the following program fragment, **wdeleteln** deletes the lines in **midscreen** until **cnt** is equal to zero:

```
int cnt;

while (cnt != 0)
{  wdeleteln(midscreen);
   cnt--;
}
```

Clearing the Window

The **wclear**, **werase**, **wclrtobot**, and **wclrtoeol** functions clear all or some of the characters from the window by replacing them with spaces. The functions prepare the window for new text.

The **wclear** function clears all characters from the window, moves the pointer to (0,0), and sets the standard screen's clear flag. The flag causes the next **wrefresh** function call to clear all characters from the window. The function call has the form:

`wclear(`*win*`)`

where *win* is the window you want to clear.

The **werase** function clears the given window, moves the pointer to (0,0), but does not set the clear flag. Use it whenever you want to preserve the contents of the terminal screen. The function call has the form:

`werase(`*win*`)`

where *win* is a pointer to the window you want to clear.

The **wclrtobot** function clears the window from the current position to the bottom of the window. The function call has the form:

`wclrtobot(`*win*`)`

where *win* is a pointer to the window you want to clear. For example, if the current position in the window **midscreen** is (10,0), the function call:

```
wclrtobot(midscreen);
```

clears all characters from line 10 and all lines below line 10.

The **wclrtoeol** function clears the window from the current position to the end of the current line. The function call has the form:

`wclrtoeol(`*win*`)`

where *win* is a pointer to the window you want to clear. For example, if the current position in **midscreen** is (10,10), the function call:

```
wclrtoeol(midscreen);
```

clears all characters from (10,10) to the end of the line. The characters at the beginning of the line remain unchanged.

The **wclrtobot** and **wclrtoeol** functions do not change the current position of the character pointer.

Refreshing from a Window

The **wrefresh** function updates the terminal screen by copying 1 or more characters from the given window to the terminal. The function effectively changes the terminal screen to reflect the new contents of the window. The function call has the form:

`wrefresh(`*win*`)`

where *win* is a pointer to a window.

Use the function solely to display changes to the window. The function copies only those characters that have changed since the last call to **wrefresh** and leaves any existing text on the terminal screen. For example, the following program fragment calls the **wrefresh** function twice:

```
waddstr(cmdmenu, "Type a command name\n");
wrefresh(cmdmenu);
waddstr(cmdmenu, "Command:  ");
wrefresh(cmdmenu);
```

In the preceding example, the first call to **wrefresh** copies the string `Type a command name` to the terminal screen. The second call copies only the string `Command:` to the terminal, because the original string has not been changed.

Note: If you give **curscr** with **wrefresh**, the function restores the screen to its most recent contents. This is useful for implementing a redraw feature for screens that become cluttered with unwanted output.

The function returns **ERR** if it encounters an error, such as illegal scrolling. If an error occurs, the function attempts to update as much of the screen as possible without causing the scroll.

Overlaying Windows

The **overlay** function copies all characters, except spaces, from one window to another, moving characters from their original positions in the first window to identical positions in the second. The function effectively lays the first window over the second, letting characters in the second window that spaces would otherwise cover remain unchanged. The function call has the form:

`overlay(`*win1*`,` *win2*`)`

where *win1* is a pointer to the window you want to copy, and *win2* is a pointer to the window to receive the copied text. The starting positions of *win1* and *win2* must match, otherwise an error occurs. If *win1* is larger than *win2*, the function copies only those lines and columns in *win1* that fit in *win2*.

The function builds a composite screen from overlapping windows. For example, in the following program fragment, **overlay** builds the standard screen from two different windows:

```
WINDOW *info, *cmdmenu;

overlay(info, stdscr);
overlay(cmdmenu, stdscr);
refresh();
```

Overwriting a Screen

The **overwrite** function copies all characters, including spaces, from their positions in the first window to identical positions in the second. The function effectively writes the contents of the first window over the second, destroying the previous contents of the second window. The function call has the form:

`overwrite(`*win1*`,` *win2*`)`

where *win1* is a pointer to the window you want to copy, and *win2* is a pointer to the window to receive the copied text. If *win1* is larger than *win2*, the function copies only those lines and columns in *win1* that fit in *win2*.

The function displays the contents of a temporary window in the middle of a larger window. For example, in the following program fragment, **overwrite** copies the contents of a work window to the standard screen:

```
WINDOW *work;

overwrite(work, stdscr);
refresh();
```

Moving a Window

The **mvwin** function moves a given window to a new position on the terminal screen, causing the upper left corner of the window to occupy a given line and column position. The function call has the form:

mvwin(*win*, *y*, *x*)

where *win* is a pointer to the window you want to move, *y* is an integer value giving the line to which you want to move the corner, and *x* is an integer value giving the column to which you want to move the corner.

The function moves a temporary window when an existing window under it contains information you want to see. For example, in the following program fragment, **mvwin** moves the window named **work** to the upper left corner of the terminal screen:

```
WINDOW *work;

mvwin(work, 0, 0);
```

The function returns **ERR** if it encounters an error, such as an attempt to move part of a window off the edge of the screen.

Reading a Character from a Window

The **inch** and **winch** functions read a single character from the current pointer position in a window or screen. The **inch** function reads a character from the standard screen. The function call has the form:

c = inch()

where *c* is the character variable to receive the character read.

The **winch** function reads a character from a given window or screen. The function call has the form:

`c = winch(`*win*`)`

where *win* is the pointer to the window containing the character to be read.

The functions compare the contents of a window with what the functions assume to be there. For example, in the following program, **inch** and **winch** compare the characters at position (0,0) in the standard screen and in the window named **altscreen**:

```
#include <stdio.h>
#include <curses.h>

WINDOW *altscreen;

main()
{  char c1, c2;

   initscr();
   altscreen = newwin(25, 80, 0, 0);
   c1 = inch();
   c2 = winch(altscreen);
   if (c1 != c2)
      error();
   endwin();
}

error()
{  printf ("Screens do not agree.\n");
   return;
}
```

Reading a character from a window does not alter the contents of the window.

Touching a Window

The **touchwin** function makes the entire contents of a given window appear to be modified, causing a subsequent **refresh** call to copy all characters in the window to the terminal screen. The function call has the form:

`touchwin(`*win*`)`

where *win* is a pointer to the window to be touched.

Use the function when two or more overlapping windows make up the terminal screen. For example, the function call:

```
touchwin(leftscreen);
```

touches the window named **leftscreen**. A subsequent **refresh** copies all characters in **leftscreen** to the terminal screen.

Deleting a Window

The **delwin** function deletes a given window from memory, freeing the space previously occupied by the window for other windows or for dynamically allocated variables. The function call has the form:

delwin(*win*)

where *win* is the pointer to the window you want to delete.

The function removes temporary windows from a program to free memory space for other uses. For example, the function call:

```
delwin(midscreen);
```

removes the window named **midscreen**.

Using Other Window Functions

The following sections explain how to perform a variety of operations on existing windows, such as setting window flags and drawing boxes around the window.

Drawing a Box

The **box** function draws a box around a window using the given characters to form the horizontal and vertical sides. The function call has the form:

box(*win, vert, hor*)

where *win* is the pointer to the desired window, *vert* is the vertical character, and *hor* is the horizontal character. Both *vert* and *hor* must have char type.

The function distinguishes one window from another when combining windows on a single screen. For example, in the following program fragment, **box** creates a box around the window in the lower half of the screen:

```
WINDOW *cmdmenu;

cmdmenu = subwin(stdscr, 5, 80, 19, 0);
box(cmdmenu, '|', '-');
```

If necessary, the function leaves the corners of the box blank to prevent illegal scrolling.

Displaying Bold Characters

The **standout** and **wstandout** functions set the standout character attribute, causing characters subsequently added to the given window or screen to appear as bold characters.

The **standout** function sets the standout attribute for characters added to the standard screen. The function call has the form:

```
standout()
```

This function call does not require arguments.

The **wstandout** function sets the standout attribute of characters added to the given window or screen. The function call has the form:

```
wstandout(win)
```

where *win* is a pointer to a window.

The functions make error messages or instructions clearly visible on the terminal screen. For example, in the following program fragment, **standout** sets the standout character attribute before adding an error message to the standard screen:

```
if (code = 5)
{
   standout();
   addstr("Illegal character.\n");
}
```

The appearance of characters with the standout attribute depends on the given terminal. The SO and SE (or US and UE) sequences define this attribute in the terminal's **termcap** entry. (For information about **termcap**, see *XENIX System Reference*).

Restoring Normal Characters

The **standend** and **wstandend** functions restore the normal character attribute, causing characters subsequently added to a given window or screen to appear as normal characters.

The **standend** function restores the normal attribute for the standard screen. The function call has the form:

```
standend()
```

This function call does not require arguments.

The **wstandend** function restores the normal attribute for a given window or screen. The function call has the form:

wstandend(*win*)

where *win* is a pointer to a window.

Use the functions after you have added an error message or instructions to a screen using the **standout** attribute. For example, in the following program fragment, **standend** restores the normal attribute after an error message has been added to the standard screen:

```
if (code == 5)
{
   standout();
   addstr("Illegal character.\n");
   standend();
}
```

Getting the Current Position

The **getyx** function copies the current line and column position of a given window pointer to a corresponding pair of variables. The function call has the form:

getyx(*win*, *y*, *x*)

where *win* is a pointer to the window containing the pointer to be examined, *y* is the integer variable to receive the line position, and *x* is the integer variable to receive the column position.

The function saves the current position so that the program can return to the position at a later time. For example, in the following program fragment, **getyx** saves the current line and column position in the variables **line** and **column**:

```
int line, column;

getyx(stdscr, line, column);
```

Setting Window Flags

The **leaveok**, **scrollok**, and **clearok** functions set or clear the cursor, scroll, and clear-screen flags. The flags control the action of the **refresh** function when called for the given window.

The **leaveok** function sets or clears the cursor flag, which defines how the **refresh** function places the terminal cursor and the window pointer after updating the screen. If the flag is set, **refresh** leaves the cursor after the last character to be copied and moves the pointer to the corresponding position in the window. If the flag is cleared, **refresh** moves the cursor to the same position on the screen as the current pointer position in the window. The function call has the form:

`leaveok(`*win*`, `*state*`)`

where *win* is a pointer to the window containing the flag, and *state* is a Boolean value defining the state of the flag. If *state* is **TRUE** the flag is set; if **FALSE**, the flag is cleared. For example, the function call:

```
leaveok(stdscr, TRUE);
```

sets the cursor flag.

The **scrollok** function sets or clears the scroll flag for the given window. If the flag is set, you can scroll through the window. If the flag is clear, you cannot scroll through the window. The function call has the form:

`scrollok(`*win*`, `*state*`)`

where *win* is a pointer to a window, and *state* is a Boolean value defining how you want to set the flag. If *state* is **TRUE**, the flag is set; if **FALSE**, the flag is cleared. The flag is initially clear, making scrolling illegal.

The **clearok** function sets and clears the clear flag for a given screen. The function call has the form:

`clearok(`*win*`, `*state*`)`

where *win* is a pointer to the desired screen, and *state* is a Boolean value. The function sets the flag if *state* is **TRUE**, and clears the flag if **FALSE**. For example, the function call:

```
clearok(stdscr, TRUE);
```

sets the clear flag for the standard screen. When the clear flag is set, each **refresh** call to the given screen automatically clears the screen by passing a clear-screen sequence to the terminal. This sequence affects the terminal only; it does not change the contents of the standard screen.

If you use **clearok** to set the clear flag for the current screen cursor, each call to refresh automatically clears the screen, regardless of the window you give in the call.

Scrolling a Window

The **scroll** function scrolls the contents of a given window upward by one line. The function call has the form:

scroll(*win*)

where *win* is a pointer to the window to be scrolled.

Combining Movement with Action

Many screen operations move the current position of a given window before performing an action on the window. For convenience, you can combine a number of functions with the movement prefix. This combination has the form:

mv*func*([*win*,] *y*, *x* [, *arg*] . . .)

where *func* is the name of a function, *win* is a pointer to the window to be operated on (**stdscr** used if you do not give one), *y* is an integer value giving the line to move to, *x* is an integer value giving the column to move to, and *arg* is a required argument for the given function. If you give more than one argument, you must separate them with commas (,). For example:

```
mvaddch(10, 5, 'X');
```

moves the position to (10,5) and adds the character X. The operation is the same as moving the position with the **move** function and then adding a character with **addch**.

For a complete list of the functions that you can use with the movement prefix in **curses**, see *XENIX System Reference*.

Controlling the Terminal

The following sections explain:

- How to set the terminal modes
- How to move the cursor
- How to access other aspects of the terminal.

Use these functions only when using other screen processing functions.

Terminal Modes

COOKED (NORAW): Normal Mode (equivalent to **stty sane**). In this mode, lines of input are collected and you can do input editing. Make the edited line available by completing it with a new-line character or by entering **Ctrl-D**. Carriage return is usually synonymous with "new line" in this mode. A new-line character replaces a carriage return whenever you type one. All tty driver functions, (input editing, interrupt generation, output processing such as delay generation and tab expansion, and so forth) are available in this mode.

CBREAK: This mode eliminates character and line editing input facilities, making the input character available to you as you type it. Flow control and interrupt and output processing are still done in this mode.

RAW: This mode eliminates all input processing and makes all input characters available as you type them. No special output processing is done.

Setting a Terminal Mode

The **crmode**, **echo**, **nl**, and **raw** functions set the terminal mode, causing subsequent input from the terminal's keyboard to be processed accordingly. These function calls do not require arguments.

The **crmode** function sets the **CBREAK** mode for the terminal. The mode preserves the function of the signal keys, allowing you to send signals to a program from the keyboard, but disables the function of the editing keys. The function call has the form:

```
crmode()
```

The **echo** function sets the **ECHO** mode for the terminal, causing each character typed at the keyboard to appear on the terminal screen. The function call has the form:

```
echo()
```

The **nl** function sets a terminal to **NEWLINE** mode, causing all new-line characters to be mapped to a corresponding new-line and return character combination. The function call has the form:

```
nl()
```

The **raw** function sets the **RAW** mode for the terminal, causing each character typed at the keyboard to be sent as direct input. The **RAW** mode disables the function of the editing and signal keys and disables the mapping of new-line characters into new-line and return combinations. The function call has the form:

```
raw()
```

Clearing a Terminal Mode

The **nocrmode**, **noecho**, **nonl**, and **noraw** functions clear the current terminal mode, allowing input to be processed according to a previous mode. These function calls do not require arguments.

The **nocrmode** function clears a terminal from the **CBREAK** mode. The function call has the form:

```
nocrmode()
```

The **noecho** function clears a terminal from the **ECHO** mode. This mode prevents characters typed at the keyboard from appearing on the terminal screen. The function call has the form:

```
noecho()
```

The **nonl** function clears a terminal from **NEWLINE** mode, causing new-line characters to be mapped into themselves. This allows the screen processing functions to perform better optimization. The function call has the form:

```
nonl()
```

The **noraw** function clears a terminal from **RAW** mode, restoring normal editing and signal generating functions to the keyboard. The function call has the form:

```
noraw()
```

Moving the Terminal's Cursor

After getting **gettmode()** and **setterm()** to get the terminal descriptions, the **mvcur** function moves the terminal's cursor from one position to another in an optimal fashion. Its usage is simple; you tell it where you are now and where you want to go. The function call has the form:

`mvcur(`*last_y*`,` *last_x*`,` *new_y*`,` *new_x*`)`

where *last_y* and *last_x* are integer values giving the last line and column position of the cursor, and *new_y* and *new_x* are integer values giving the new line and column position of the cursor. For example, the function call:

```
mvcur(10, 5, 3, 0);
```

moves the cursor from (10, 5) to (3, 0) on the terminal screen. The function call:

```
mvcur(0, 0, LINES/2, COLS/2)
```

moves the cursor from the home position (0, 0) to the middle of the screen.

Note: Use the **mvcur** function only in programs that do not use other screen processing functions. This means that you can use the function to perform optimal cursor motion witheout the aid of the other functions. For programs that do use other functions, you must use the **move**, **wmove**, **refresh**, and **wrefresh** functions to move the cursor.

Getting the Terminal Mode

The **gettmode** function returns the current terminal mode. The function call has the form:

`s = gettmode()`

where *s* is the variable to receive the status.

The **initscr** function normally calls this function.

Variables Set by gettmode ()

If US and UE do not exist in the **termcap** entry, they are copied from SO and SE in **setterm()**.

Type	Name	Description
bool	NONL	Terminal cannot handle line feeds during a CR
bool	GT	**gtty** indicates tabs
bool	UPPERCASE	Terminal generates only uppercase letters.

Saving and Restoring the Terminal Flags

The **savetty** function saves the current terminal flags, and the **resetty** function restores the flags previously saved by the **savetty** function. The **initscr** and **endwin** functions automatically perform the **savetty** and **resetty** functions. You do not need them when performing ordinary screen processing.

Setting a Terminal Type

The **setterm** function sets the terminal type to the given type. The function call has the form:

setterm(*name*)

where *name* is a pointer to a string containing the terminal type identifier. The **initscr** function normally calls this function. You can also call **setterm** directly.

Variables Set by setterm()

Type	Name	Pad	Description
char *	al	Pn[1]	Add new blank line
bool	am		Automatic margins
char *	bc		Back cursor movement
bool	bs		Backspace enabled
char *	bt	P	Back tab

---

[1] P indicates that padding may be specified. Pn indicates that padding may be based on the number of lines affected.

Type	Name	Pad	Description
bool	ca		Cursor addressable
char *	cd	Pn	Clear to end of display
char *	ce	P	Clear to end of line
char *	cl	Pn	Clear screen
char *	cm	P	Cursor motion
char *	cr	Pn	Carriage return
bool	da		Display may be retained above
bool	db		Display may be retained below
char *	dc	Pn	Delete character
char *	dl	Pn	Delete line sequence
char *	dm		Delete mode (enter)
char *	do		Down line sequence
char *	ed		End delete mode
bool	eo		Can erase overstrikes with ' '
char *	ei		End insert mode
char *	ho		Home cursor
bool	hz		Hazeltine terminal
char *	ic	P	Insert character
char *	im		Enter insert mode (IC usually set, too)
bool	in		Insert mode distinguishes nulls on display
char *	ip	Pn	Pad after character inserted using IM + EI
char *	ll		Quick to last line, column 0
char *	ma		Ctrl character map for cmd mode
bool	mi		Can move in insert mode
bool	nc		No CR: \r sends \r\n then deletes \n
char *	nd		Nondestructive space
char *	nl	Pn	Newline character
bool	os		Overstrike enabled
char	pc		Pad character
char *	se		Standout end (may leave space)
char *	sf	P	Scrolls forward
char *	so		Standout begin (may leave space)
char *	sr	P	Scrolls in reverse
char *	ta	P	Tab (not ^ I or with padding)
char *	te		Terminal address enable ending sequence
char *	ti		Terminal address enable initialization
char *	uc		Underline a single character
char *	ue		Underline ending sequence
bool	ul		Terminal underlines even though it does not overstrike
char *	up		Upline

Type	**Name**	**Pad**	**Description**
char *	us		Underline starting sequence
char *	vb		Visible bell
char *	ve		Visual end sequence
char *	vs		Visual start sequence
bool	xn		A newline is lost after wrap

For a complete description and terminal capabilities list, see **termcap** in *XENIX System Reference.*

Reading the Terminal Name

The **longname** function converts a given **termcap** identifier into the full name of the corresponding terminal. The function call has the form:

`longname(`*termbuf, name*`)`

where *termbuf* is a pointer to the string containing the terminal type identifier, and *name* is a character pointer to the location to receive the long name. The terminal type identifier must exist in the **/etc/termcap** file.

The function gets the full name of the terminal you are using. The variable **ttytype** stores the current terminal's identifier; **ttytype** also can receive a new name.

Chapter 5. Character and String Processing

Character and string processing is an important part of many programs. Programs regularly assign, manipulate, and compare characters and strings to complete their tasks. For this reason, the standard library provides a variety of character and string processing functions. These functions give a convenient way to test, translate, assign, and compare characters and strings.

To use the character functions in a program, you must include the file **ctype.h**, which provides the definitions for special character macros, in the program. The line:

```
#include <ctype.h>
```

must appear at the beginning of the program.

You do not need to use any command to use the string functions. The standard C library defines these functions and they are read whenever you compile a C program.

Using the Character Functions

The character functions test and convert characters. Many character functions are defined as macros and, as such, you cannot redefine them or use them as a target for a breakpoint when debugging.

Testing for an ASCII Character

The **isascii** function tests for characters in the ASCII character set, that is, characters whose values range from 0 to 127. The function call has the form:

```
isascii(c)
```

where *c* is the character you want to test. The function returns a nonzero (true) value if the character is ASCII, otherwise it returns zero (false). For example, in the following program fragment, **isascii** is used to determine whether the value in *c* read from the file given by **data** is in the acceptable ASCII range:

```
FILE *data;
int c = fgetc(data);

if (!isascii(c))
   notext();
```

In this example, a function named **notext** is called if the character is not in range.

Converting to ASCII Characters

The **toascii** function converts non-ASCII characters to ASCII. The function call has the form:

```
c = toascii(i)
```

where *c* is the variable to receive the character, and *i* is the value you want to change. The function creates an ASCII character by truncating all but the low-order 7 bits of the non-ASCII value. If the *i* value is already an ASCII character, no change takes place. For example, the function call:

```
ascii = toascii(160);
```

converts value 160 to 32, the ASCII value of the space character.

The function prepares non-ASCII characters for display at the standard output. For example, in the following program fragment, **toascii** converts each character read from the file given by **oddstrm**:

```
FILE *oddstrm;
int c = toascii(getc(oddstrm));

if (isprint(c) || isspace(c))
   putchar(c);
```

If the resulting character is printable or is whitespace, it is written to the standard output.

Testing for Alphanumerics

The **isalnum** function tests for letters and decimal digits, that is, the alphanumeric characters. The function call has the form:

```
isalnum(c)
```

where *c* is the character to test. The function returns a nonzero (true) value if the character is an alphanumeric; otherwise, it returns zero (false). For example, the function call:

```
isalnum('1');
```

returns a nonzero value, but the call:

```
isalnum('>');
```

returns zero.

Testing for a Letter

The **isalpha** function tests for uppercase or lowercase letters, that is, alphabetic characters. The function call has the form:

```
isalpha(c)
```

where *c* is the character you want to test. The function returns a nonzero (true) value if the character is a letter; otherwise, it returns zero. For example, the function call:

```
isalpha('a');
```

returns a nonzero value, but the call:

```
isalpha('1');
```

returns zero.

Testing for Control Characters

The **iscntrl** function tests for control characters, that is, characters whose ASCII values are in the range 0 to 31 and 127. The function call has the form:

```
iscntrl(c)
```

where *c* is the character you want to test. The function returns a nonzero (true) value if the character is a control character, otherwise it returns zero (false). For example, in the program following fragment, **iscntrl** determines whether or not the character in *c* read from the file given by **infile** is a control character:

```
FILE *infile, *outfile;
int c = fgetc(infile);

if (!iscntrl(c))
   fputc(c, outfile);
```

The **fputc** function is not executed if the character is a control character.

Testing for a Decimal Digit

The **isdigit** function tests for decimal digits. The function call has the form:

```
isdigit(c)
```

where *c* is the character you want to test. The function returns a nonzero value if the character is a digit; otherwise, it returns zero. For example, the following program fragment performs translation from ASCII decimal to internal binary (similar to the task that **atoi()** performs): digit:

```
FILE *infile;
int c, num = 0;

while (isdigit(c = getc(infile)))
   num = num * 10 + c - '0';
```

Testing for a Hexadecimal Digit

The **isxdigit** function tests for a hexadecimal digit. The hexadecimal digits are the uppercase letters **A** through **F**, the lowercase letters **a** through **f**, and the numbers **0** (zero) through **9**. The **isxdigit** function call has the form:

```
isxdigit(c)
```

where *c* is the character you want to test. The function returns a nonzero value if the character is a hexadecimal digit; otherwise, it returns zero. For example, in the following program fragment, **isxdigit** tests whether a hexadecimal digit is read from the standard input:

```
int c = getchar();

if (isxdigit(c))
   hexmode();
```

In this example, a function named **hexmode** is called if a hexadecimal digit is read.

Testing for Printable Characters

The **isprint** function tests for printable characters, that is, characters whose ASCII values range from 32 to 126. The function call has the form:

`isprint(c)`

where *c* is the character you want to test. The function returns a nonzero value if the character is printable; otherwise, it returns zero.

Testing for Punctuation

The **ispunct** function tests for punctuation characters, that is, characters that are neither control characters nor alphanumeric characters. The function call has the form:

`ispunct(c)`

where *c* is the character you want to test. The function returns a nonzero function if the character is a punctuation character; otherwise, it returns zero.

Testing for Whitespace

The **isspace** function tests for whitespace characters, that is, the space, horizontal tab, vertical tab, form feed, and new-line characters. The function call has the form:

`isspace(c)`

where *c* is the character you want to test. The function returns a nonzero value if the character is a whitespace character, otherwise it returns zero.

Testing for Case in Letters

The **isupper** and **islower** functions test for uppercase and lowercase letters, respectively. The function calls have the forms:

```
isupper(c)
```

and

```
islower(c)
```

where *c* is the character you want to test. The function returns a nonzero value if the character is the proper case, otherwise it returns zero. For example, the function call:

```
isupper('b');
```

returns zero (false), but the call:

```
islower('b');
```

returns a nonzero (true) value.

Converting the Case of a Letter

The **tolower** and **toupper** functions convert the case of a given letter. The function calls have the forms:

```
c = tolower(i)
```

and

```
c = toupper(i)
```

where *c* is the variable to receive the converted letter, and *i* is the letter you want to convert. For example, the function call:

```
lower = tolower('B');
```

converts **B** to **b** and assigns it to **lower**, and the call:

```
upper = toupper('b');
```

converts **b** to **B** and assigns it to **upper**.

The **tolower** function returns the character unchanged if it is not an uppercase letter. Similarly, the **toupper** function returns the character unchanged if it is not a lowercase letter.

These functions make the case of the characters read from a file or standard input consistent. For example, in the following statement, **tolower** changes the character read from the standard input to lowercase before it is compared:

```
if (tolower(getchar()) != 'y')
    exit(0);
```

This conversion allows the user to type either **Y** or **y** to prevent the statement from executing the **exit** function.

Using the String Functions

The string functions concatenate, compare, copy, and count the characters in a string. Two special string functions, **sscanf** and **skrintf**, let a program read from and write to a string in the same way the standard input and output can be read and written. These functions are convenient when one reads or writes whole lines containing values of several formats.

Many string functions have two forms: a form that manipulates all characters in the string and one that manipulates a given number of characters. This gives programs control over all or parts of strings.

Concatenating Strings

The **strcat** function concatenates two strings by appending the characters of one string to the end of another. The function call has the form:

`strcat(`*dst*`,` *src*`)`

where *dst* is a pointer to the string to receive the new characters, and *src* is a pointer to the string containing the new characters. The function appends the new characters in the same order as they appear in *src*, then appends a null character (\0) to the last character in the new string. Make the receiving string large enough to contain the new string and its trailing null character (\0). The function cannot detect overflow. The function always returns the pointer *dst*.

The function can build a string such as a full path name from two smaller strings. For example, in the following program fragment **strcat** concatenates the string **temp** to the contents of the character array **dir**:

```
#define MAX 25

char dir[MAX] = "/usr/";

main()
{
   strcat(dir, "temp");
}
```

Comparing Strings

The **strcmp** function compares the characters in one string to those in another and returns an integer value showing the result of the comparison. The function call has the form:

`strcmp(s1, s2)`

where *s1* and *s2* are the pointers to the strings you want to compare. The function returns zero if the strings are equal (that is, have the same characters in the same order). If *s1* is lexicographically greater than *s2*, **strcmp** returns an integer value greater than **0** (zero). Otherwise, *s1* is lexicographically less than *s2*, and **strcmp** returns a integer value less than **0** (zero). For example, the function call:

```
strcmp("Character A", "Character A");
```

returns zero because the strings are identical in every way. The function call:

```
strcmp("Character A", "Character B");
```

returns -1 because the ASCII value of B is one greater than A.

The **strcmp** function continues to compare characters until it finds a mismatch. If one string is shorter than the other, the function stops at the end of the shorter string. For example, the function call:

```
strcmp("Character A", "Character");
```

returns 32, that is, the difference between the space character and the null character.

Copying a String

The **strcpy** function copies a given string to a given location. The function call has the form:

```
strcpy(dst, src)
```

where *dst* is a pointer to the location to receive the string, and *src* is a pointer to the string **strcpy** copies. The function copies all characters in the source string *src* to *dst* and appends a null character (\0) to the end of the new string. If *dst* contained a string before the copy, that string is destroyed. The function always returns the pointer to the new string.

For example, in the following program fragment, **strcpy** copies the string "not available" to the location given by **name**:

```
char na[ ] = "not available";
char name[20];

main()
{
   strcpy(name, na);
}
```

The location to receive a string must be large enough to contain the string. The function cannot detect overflow.

Getting the Length of a String

The **strlen** function returns the number of characters in a given string. The function call has the form:

```
strlen(s)
```

where *s* is a pointer to a string. The count includes all characters up to, but not including, the first null character. The return value is always an integer.

In the following program fragment, **strlen** determines whether the contents of **inname** are short enough for **name** to store them:

```
char *inname;
char name[MAX];

if (strlen(inname) < MAX)
   strcpy(name, inname);
```

Concatenating Characters to a String

The **strncat** function appends one or more characters to the end of a given string. The function call has the form:

`strncat(`*dst*`, `*src*`, `*n*`)`

where *dst* is a pointer to the string to receive the new characters, *src* is a pointer to the string containing the new characters, and *n* is an integer value giving the number of characters you want to concatenate. Make the receiving string large enough to contain the new string. The function cannot detect overflow. The function appends the given number of characters to the end of the *dst* string, then returns the pointer *dst*.

In the following program fragment, **strncat** copies the first 3 characters in **letter** to the end of **cover**:

```
char cover[9] = "cover";
char letter[ ] = "letter";

main()
{
   strncat(cover, letter, 3);
}
```

This example creates the new string **coverlet** in **cover**.

Comparing Characters in Strings

The **strncmp** function compares one or more pairs of characters in two given strings and returns an integer value that gives the result of the comparison. The function call has the form:

`strncmp(`*s1*`, `*s2*`, `*n*`)`

where *s1* and *s2* are pointers to the strings to be compared, and *n* is an integer value giving the number of characters to compare. The function returns zero if the first *n* characters are identical.

If the first *n* characters in *s1* are lexicographically greater than the first *n* characters in *s2*, **strncmp** returns an integer value greater than **0** (zero). Otherwise, the first *n* characters in *s1* are lexicographically less than the first *n* characters in *s2*, and **strncmp** returns a integer value less than **0** (zero). The function generates the difference by subtracting the mismatched character of the second string from the character in the first string.

For example, the function call:

```
strncmp("Character A", "Character B", 5);
```

returns zero because the first 5 characters are identical. The function call:

```
strncmp("Character A", "Character B", 11);
```

returns -1 because the value of B is one greater than A.

The function continues to compare characters until it finds a mismatch or the end of a string.

Copying Characters to a String

The **strncpy** function copies a given number of characters to a given string. The function call has the form:

`strncpy(`*dst*`,` *src*`,` *n*`)`

where *dst* is a pointer to the string to receive the characters, *src* is a pointer to the string containing the characters, and *n* is an integer value giving the number of characters you want to copy. The function copies either the first *n* characters in *src* to *dst*, or if *src* has fewer than *n* characters, copies all characters up to the first null character. Make the receiving string large enough to contain the new string. The function cannot detect overflow. The function returns the address of *dst*.

In the following program fragment, **strncpy** copies the first three characters in `date` to `day`:

```
#include <stdio.h>

#define MAX 25

char date[] = "Fri Dec 29 09:35:44 EDT 1982";

main()
{
   char day[MAX];

   strncpy(day, date, 3);
   day[3] = '\0';
   printf("day = %s\n", day);
}
```

In this example, `day` receives the string `"Fri"`. The null character (`'\0'`) is not appended to the string `"Fri"`. Therefore, the user must append the null character.

Reading Values from a String

The **sscanf** function reads one or more values from a given character string and stores the values at a given memory location. The function is similar to the **scanf** function, which reads values from the standard input. The function call has the form:

```
sscanf(s, format, argptr  . . . )
```

where *s* is a pointer to the string to be read, *format* is a pointer to the string defining the format of the values to be read, and *argptr* is a pointer to the variable that is to receive the values read. If you give more than one *argptr*, you must separate them with commas. The *format* string may contain the same formats as given for **scanf** (see **scanf** in *XENIX System Reference*). The function always returns the number of values read.

The function reads values from a string containing several values of different formats, or reads values from a program's own input buffer. For example, in the following program fragment, **sscanf** reads two values from the string pointed to by **datestr**:

```
char datestr[ ] = {"THU MAR 29 11:04:40 EST 1983"};
char month[4];
char year[5];

main()
{
   sscanf(datestr, "%*3s%3s%*2s%*8s%*3s%4s", month, year);
   printf("%s, %s\n", month, year);
}
```

The location pointed to by **month** stores the first value (a 3-character string). The location pointed to by **year** stores the second value (a 4-character string).

Writing Values to a String

The **skrintf** function writes one or more values to a given string. The function call has the form:

```
sprintf(s, format [,arg] . . . )
```

where *s* is a pointer to the string to receive the value, *format* is a pointer to a string that defines the format of the values to be written, and *arg* is the variable or value to be written. If you give more than one *arg*, you must separate them with commas (,). The *format* string may contain the same formats as given for **printf** (see **printf** in *XENIX System Reference*). After the function writes all values to the string, the function adds a null character (**0**) to the end of the string. The function normally returns the number of characters processed, but returns a negative value if an output error was encountered.

The function builds a large string from several values of different formats. For example, in the following program fragment, **skrintf** writes three values to the string pointed to by **cmd**:

```
char cmd[100];
char *doc = "/usr/src/cmd/cp.c";

int width = 50;
int length = 60;

sprintf(cmd, "pr -w%d -l%d %s\n", width, length, doc);
system(cmd);
```

In this example, a call to the **system** function uses the string that the **skrintf** function creates. The first two values are the decimal numbers given by width and length. The last value is a string (a file name) to which **doc** points. The final string has the form:

```
pr -w50 -l60 /usr/src/cmd/cp.c
```

The string to receive the values must have sufficient length to store those values. The function cannot check for overflow.

Chapter 6. Using Process Control

This chapter describes the process control functions of the standard C library. The functions let a program call other programs, using a method similar to calling functions.

There are a variety of process control functions. The **system** and **exit** functions provide the highest level of execution control. Most programs that need a straightforward way to call another program or terminate the current one use these functions. The **execl**, **execv**, **fork**, and **wait** functions provide low-level control of execution and are for programs that must have very fine control over their own execution and the execution of other programs. Other process control functions, such as **abort** and **exec**, are described in detail in *XENIX System Reference.*

The process control functions are a part of the standard C library. Because this library is automatically read when you compile a C program, you do not need any special library argument when you invoke the compiler.

Using Processes

The term ***process*** describes a program that PC XENIX executes. A process consists of instructions and data and a table of information about the program, such as its allocated memory, open files, and current execution status.

You create a process whenever you invoke a program through a shell. The system assigns a unique process ID to a program when you invoke it, and uses this ID to control and manage the program. A system running several processes at the same time needs the unique process IDs.

You can also create a process by directing a program to call another program. This causes the system to perform the same functions as when it invokes a program through a shell. In fact, these two methods are actually the same method; invoking a program through a shell is nothing more than directing a program (the shell) to call another program.

The system handles all processes in essentially the same way, so the sections that follow should give you valuable information for writing your own programs and an insight into PC XENIX.

Calling a Program

The **system** function calls the given program, executes it, and then returns control to the original program. The function call has the form:

system(*command-line*)

where *command-line* is a pointer to a string containing a shell command line. The command line must begin with the program name followed by any required or optional arguments. For example, the call:

```
system("date");
```

causes the system to execute the **date** command, which displays the current time and date at the standard output. The call:

```
system("cat >response");
```

causes the system to execute the **cat** command. In this case, the standard output is redirected to the file **response**, so the command reads from the standard input and copies this input to the file **response**.

The **system** function operates the same way as a function call to execute a program and return to the original program. For example, in the following program fragment, **system** calls a program whose name you give in the string **cmd**:

```
char name[40], cmd[50];

printf("Enter filename:  ");
scanf("%s", name);
sprintf(cmd, "cat %s", name);
system(cmd);
```

The **skrintf** function builds the string in **cmd**. This string contains the program name **cat** and an argument (the file name read by **scanf**). The effect is to execute the **cat** command with the given file name.

PC XENIX uses buffered input. If a program uses the **getc** or the **putc** function, the program must read all of the input or output, or flush out the remaining input or output before calling the

system function. (You can do this for output with the **fflush** function and for input with the **setbuf** function described in "Using More Stream Functions" on page 3-24.) Otherwise, the **system** function may receive the input that was intended for the calling program.

Stopping a Program

The **exit** function stops the execution of a program by returning control to the system. The function call has the form:

`exit(`*status*`)`

where *status* is the integer value you want to send to the system as the termination status.

The function terminates a program before its normal end, such as after a serious error. For example, in the following program fragment, **exit** stops the program and sends the integer value 2 to the system if the **fopen** function returns the null pointer value **NULL**:

```
if (fopen("myfile", "r") == NULL)
   exit(2);
```

The **exit** function automatically closes each open file in the program before returning to the system. This means you do not need to call the **fclose** or **close** functions before an exit.

Starting a New Program

The **execl** and **execv** functions cause the system to overlay the calling program with the given one, allowing the calling program to terminate while the new program continues execution.

The **execl** function call has the form:

`execl(`*pathname*`, `*command-name*`, `*argptr*` . . . )`

where *pathname* is a pointer to a string containing the full path name of the command you want to execute, *command-name* is a pointer to a string containing the name of the program you want to execute, and *argptr* is one or more pointers to strings that contain the program arguments. You must separate each *argptr* from any other argument by a comma. The last *argptr* in the list must be the null pointer value **NULL**. For example, in the call:

```
execl("/bin/date", "date", NULL);
```

the **date** command, whose full path name is **/bin/date**, takes no arguments. In the call:

```
execl("/bin/cat", "cat", file1, file2, NULL);
```

the **cat** command, whose full path name is **/bin/cat**, takes the pointers file1 and file2 as arguments.

The **execv** function call has the form:

execv(*pathname*, *ptr*);

where *pathname* is the full path name of the program you want to execute, and *ptr* is pointer to an array of pointers. Each element in the array must point to a string. The array may have any number of elements, but the first element must point to a string containing the program name, and the last must be the null pointer, **NULL**.

Use the **execl** and **execv** functions in programs that execute in two or more phases and communicate through temporary files (for example a two-pass compiler). The first part of such a program calls the second part by giving the name of the second part and the appropriate arguments. For example, the following program fragment checks the status of **errflag**, then either overlays the current program with the program **pass2**, or displays an error message and quits:

```
char *tmpfile = "tmpfile";
int errflag;

if (errflag == 0)
   execl("/usr/bin/pass2", "pass2", tmpfile, NULL);
else
{
   fprintf(stderr, "Error %d:  Quitting", errflag);
   exit(2);
}
```

The **execv** function passes arguments to a program when the precise number of arguments is not known beforehand. For example, the following program fragment reads arguments from the command line (beginning with the third one), copies the pointer of each to an element in **cmd**, sets the last element in **cmd** to **NULL**, and executes the **cat** command:

```
#include <stdio.h>

#define MAX 64

main(argc, argv)
int argc;
char *argv[];
{
   int i = 1;
   char *cmd[MAX];

   cmd[0] = "cat";
   while (i<argc && i<MAX-1)
   {
      cmd[i] = argv[i];
      ++i;
   }
   cmd[i] = NULL;
   execv("/bin/cat", cmd);
}
```

The **execl** and **execv** functions return control to the original program only if there is an error in finding the given program (for example, a misspelled path name or no execute permission). This allows the original program to check for errors and display an error message if necessary. For example, the following program fragment searches for the program **display** in the **/usr/bin** directory:

```
execl("/usr/bin/display", "display", NULL);
fprintf(stderr, "Can't execute 'display' \n");
```

If the program **display** is not found or lacks the necessary permissions, the original program resumes control and displays an error message.

The **execl** and **execv** functions do not expand metacharacters (for example, <, >, *, ?, and [ ]) given in the argument list. If a program needs these features, it can use **execl** or **execv** to call a shell as described in the next section.

Executing a Program Through a Shell

One drawback of the **execl** and **execv** functions is that they do not provide the metacharacter features of a shell. One way to overcome this problem is to use **execl** to execute a shell and let the shell execute the command you want.

The function call has the form:

execl("/bin/sh", "sh", "-c", *command-line*, NULL);

where *command-line* is a pointer to the string containing the command line needed to execute the program. The string must be exactly as it would appear if typed at the terminal.

For example, a program can execute the command:

```
cat *.c
```

(which contains the metacharacter *) with the call:

```
execl("/bin/sh", "sh", "-c", "cat *.c", NULL);
```

In this example, the full path name **/bin/sh** and command name **sh** start the shell. The argument **-c** causes the shell to treat the argument **cat *.c** as a whole command line. The shell expands the metacharacter and displays all files that end with **.c**, something that the **cat** command cannot do by itself.

Duplicating a Process

The **fork** function splits an executing program into two independent and fully functioning processes. The function call has the form:

```
fork()
```

This function call does not require arguments.

The function makes several copies of any program that must take divergent actions as a part of its normal operation, for example, a program that must use the **execl** function yet still continue to execute. The original program, called the parent process, continues to execute normally, just as it would after any other function call. The new process, called the child process, starts its execution at the same point, that is, just after the **fork** call. (The child never goes back to the beginning of the program to start execution.) The two processes are in effect synchronized and continue to execute as independent programs.

The **fork** function returns a different value to each process. To the parent process, the function returns the process ID of the child. The process ID is always a positive integer and is always different from the parent's ID. To the child, the function returns 0. All other variables and values remain exactly as they were in the parent.

The return value determines which steps the child and parent should take next. For example, in the program fragment:

```
char *command = "cat *.c";

if (fork() == 0)
    execl("/bin/sh", "sh", "-c", command, NULL);
```

the child's return value, 0, causes the expression **fork() = = 0** to be true, and therefore the **execl** function is called. The parent's return value, on the other hand, causes the expression to be false, and the function call is skipped. Executing the **execl** function causes the program given by *command* to overlay the child. This does not affect the parent.

If **fork** encounters an error and cannot create a child, it returns the value -1. It is a good idea to check for this value after each call.

Waiting for a Process

The **wait** function causes a parent process to wait until its child processes have completed their execution before continuing its own execution. The function call has the form:

`wait(`*ptr*`)`

where *ptr* is a pointer to an integer variable. It receives the termination status of the child from both the system and the child itself. The function normally returns the process ID of the terminated child, so the parent may check it against the value returned by **fork**.

The function synchronizes the execution of a parent and its child and is useful if the parent and child processes access the same files.

For example, the following program fragment causes the parent to wait while the program named by *pathname* (which has overlaid the child process) finishes its execution:

```
int status;
char *pathname;
char *cmd [ ];

if (fork() == 0)
   execv(pathname, cmd);
wait(&status);
```

The **wait** function always copies a status value to its argument. The status value is two 8-bit values combined into one. The low-order 8 bits are the termination status of the child as defined by the system. This status is zero for normal termination and nonzero for other kinds of termination, such as termination by an interrupt, quit, or hangup signal. The next 8 bits are the termination status of the child as defined by its own call to **exit**. If the child did not explicitly call the function, the status is zero.

Inheriting Open Files

Any program called by another program or created as a child process to a program automatically inherits the original program's open files and standard input, output, and error files. This means if the file was open in the original program, it is open in the new program or process.

A new program also inherits the contents of the input and output buffers used by the open files of the original program. To prevent a new program or process from reading or writing data not intended for its use, these buffers should be flushed before calling the program or creating the new process. A program can flush an output buffer with the **fflush** function and an input buffer with **setbuf**.

Program Example

This section shows how to use the process control functions to control a simple process. The following program starts a shell on the terminal given in the command line. The terminal is assumed to be connected to the system through a line that has not been enabled for multiuser operation:

```
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[ ];
{
   int status;

   if (argc < 2)
   {
      fprintf(stderr, "No tty given.\n");
      exit(1);
   }
   if (fork() == 0)
   {
      if (freopen(argv[1], "r", stdin) == NULL)
         exit(2);
      if (freopen(argv[1], "w", stdout) == NULL)
         exit(2);
      if (freopen(argv[1], "a", stderr) == NULL)
         exit(2);
      execl("/bin/sh","sh",NULL);
   }
   wait(&status);
   if (status == 512)
      fprintf(stderr, "Bad tty name:  %s\n", argv[1]);
}
```

In this example, the **fork** function creates a duplicate copy of the program. The child changes the standard input, output, and error files to the new terminal by closing and reopening them with the **freopen** function. The terminal name pointed to by **argv** must be the name of the device special file associated with the terminal, for example, **/dev/tty01**. The **execl** function then calls the shell, which uses the new terminal as its standard input, output, and error files.

The parent process waits for the child to terminate. The **exit** function terminates the process if an error occurs when reopening the standard files. Otherwise, the process continues until the Ctrl-D key is pressed at the new terminal.

Chapter 7. Creating and Using Pipes

A pipe is a way a program can pass information to other programs without any temporary file. A pipe connects the output of one program to the input of another program. A pipe is similar to a file in that it has a file pointer and/or a file descriptor and can be read from or written to using the input and output functions of the standard library. Unlike a file, a pipe does not represent a specific file or device. Instead, a pipe represents temporary storage in memory that is independent of the program's own memory and is controlled entirely by the operating system.

Pipes are chiefly used to pass information among programs, just as the shell pipe symbol (¦) is used to pass the output of one program to the input of another. This eliminates the need to create temporary files to pass information to other programs. A pipe can also be used as a temporary storage place for a single program. A program can write to the pipe, then read that information back at a later time.

The standard library provides several pipe functions. The **popen** and **pclose** functions control both a pipe and a process. The **popen** function opens a pipe and creates a new process at the same time, making the new pipe the standard input or output of the new process. The **pclose** function closes the pipe and waits for termination of the corresponding process. The **pipe** function, on the other hand, gives low-level access to a pipe. The function is similar to the **open** function, but opens the pipe for both reading and writing, returning two file descriptors instead of one. The program can either use both sides of the pipe or close the side it does not need. The low-level input and output functions **read** and **write** can be used to read from and write to a pipe. Pipe file descriptors are used in the same way as other file descriptors.

Opening a Pipe to a New Process

The **popen** function creates a new process and then opens a pipe to the standard input or output file of that new process. The function call has the form:

popen(*command*, *type*)

where *command* is a pointer to a string that contains a shell command line, and *type* is a pointer to the string that defines whether the original process will open the pipe for reading or writing. It may be **r** for reading or **w** for writing. The function normally returns the file pointer to the open pipe, but returns the null pointer value **NULL** if an error is encountered.

The function is used in programs that call another program and pass substantial amounts of data to that program. For example, in the following program fragment, **popen** creates a new process for the **cat** command and opens a pipe for writing to that command:

```
FILE *pstrm;

pstrm = popen("cat >response", "w");
fprintf(pstrm, "hello, cat . . . \n");
```

The new pipe given by **pstrm** links the standard input of the command with the program. Data written to the pipe is used as input by the **cat** command. `"hello, cat . . . \n"` is written to the file `response`.

Reading and Writing to a Process

The **fscanf**, **fprintf**, and other stream functions can read from or write to a pipe opened by the **popen** function.

The **fscanf** function can read from a pipe opened for reading. For example, in the following program fragment, **fscanf** reads from the pipe given by **pstrm**:

```
FILE *pstrm;
char name[20];
int number;

pstrm = popen("cat list.c","r");
fscanf(pstrm, "%s %d", name, &number);
```

This pipe is connected to the standard output of the **cat** command, so **fscanf** reads the first name and number written by **cat** to its standard output.

The **fprintf** function can write to a pipe opened for writing. For example, in the following program fragment, **fprintf** writes the string pointed to by **buf** to the pipe given by **pstrm**:

```
FILE *pstrm;
char *buf = "a string";

pstrm = popen("wc", "w");
fprintf(pstrm, "%s", buf);
```

This pipe is connected to the standard input of the **wc** command, so the command reads and counts the contents of **buf**.

Closing a Pipe

The **pclose** function closes the pipe opened by the **popen** function. The function call has the form:

`pclose(`*stream*`)`

where *stream* is the file pointer of the pipe you want to close. The function normally returns the exit status of the command that was issued as the first argument of its corresponding **popen**, but returns the value -1 if **popen** did not open the pipe.

For example, in the following program fragment, **pclose** closes the pipe given by **pstrm** if the end-of-file value **EOF** has been found in the pipe:

```
FILE *pstrm;

if (feof(pstrm))
   pclose(pstrm);
```

Opening a Low-Level Pipe

The **pipe** function opens a pipe for both reading and writing. The function call has the form:

`pipe(`*fd*`)`

where *fd* is a pointer to a 2-element array. It must have **int** type. Each element receives one file descriptor. The first element receives the file descriptor for the reading side of the pipe, and the other element receives the file descriptor for the writing side. The function normally returns 0, but returns the value -1 if it encounters an error. For example, in the following program fragment, **pipe** creates two file descriptors if it does not encounter an error:

```
int chan[2];

if (pipe(chan) == -1)
   exit(2);
```

The array element chan[0] receives the file descriptor for the reading side of the pipe, and chan[1] receives it for the writing side.

The function opens a pipe in preparation for linking it to a child process. For example, in the following program fragment, **pipe** causes the program to create a child process if it successfully creates a pipe:

```
int fd[2];

if (pipe(fd) != -1)
   if (fork() == 0)
      close(fd[1]);
```

The child process closes the writing side of the pipe. The parent can now pass data to the child by writing to the pipe; the child can retrieve the data by reading the pipe.

Reading and Writing to a Low-Level Pipe

The **read** and **write** input and output functions can read and write characters to a low-level pipe.

The **read** function can read from the read side of an open pipe. For example, in the following program fragment, **read** reads **MAX** characters from the read side of the pipe given by **chan**:

```
int chan[2];
char buf[MAX];
int number;

pipe(chan);
number = read(chan[0], buf, MAX);
```

In this example, **read** stores the characters in the array **buf**.

Unless it encounters the end-of-file character, a **read** call waits for the given number of characters to be read before returning.

The **write** function can write to the write side of a pipe. For example, in the following program fragment, **write** writes **MAX** characters from the character array **buf** to the writing side of the pipe given by **chan**:

```
int chan[2];
char buf[MAX];
int number;

pipe(chan);
number = write(chan[1], buf, MAX);
```

If the **write** function finds that a pipe is full, it waits until some characters have been read before completing its operation.

Closing a Low-Level Pipe

The **close** function can be used to close the reading or the writing side of a pipe. For example, the function call:

```
close(chan[0]);
```

closes the reading side of the pipe given by **chan**, and the call:

```
close(chan[1]);
```

closes the writing side.

The system copies the end-of-file value **EOF** to a pipe when the process that made the original pipe, and every process created or called by that process, has closed the writing side of the pipe. This means, for example, that if a parent process is sending data to a child process through a pipe and closes the pipe to signal the end of the file, the child process will not receive the end-of-file value unless it has already closed its own write side of the pipe.

Program Examples

This section shows how to use the process control functions with the low-level **pipe** function to create functions similar to the **popen** and **pclose** functions.

The first example is a modified version of the **popen** function. The modified function identifies the new pipe with a file descriptor rather than a file pointer. It also requires a mode argument rather than a type argument, where the mode is 0 for reading or 1 for writing:

```
#include <stdio.h>
#define READ  0
#define WRITE 1

xpopen(cmd, mode)          /* "fd" version of popen() */
char *cmd;                 /* command -- "cat", etc   */
int mode;                  /* READ or WRITE           */
{  int fd[2];
   int pid;

   if (mode < 0 || mode > 1)  /* invalid mode?       */
      return(-1);
   if (pipe(fd) == -1)         /* failed pipe?       */
      return(-1);
   if ((pid = fork()) == -1)   /* failed fork?       */
      return(-1);
   if (pid == 0)               /* child process      */
   {  if (mode == READ)        /* parent <- child    */
      {  close(1);             /* close stdout       */
         if (dup(fd[1]) == -1) /* stdout = fd[1]     */
             return(-1);
      }
      else                     /* parent -> child    */
      {  close(0);             /* close stdin        */
         if (dup(fd[0]) == -1) /* stdin = fd[0]      */
            return(-1);
      }
      close(fd[0]);            /* will not be used   */
      close(fd[1]);            /* will not be used   */
      execl("/bin/sh", "sh","-c", cmd, NULL);
      exit(1);                 /* should not execute */
   }
   else                        /* parent process     */
   {  if (mode == READ)        /* parent <- child    */
      {  close(fd[1]);         /* will not be used   */
         return(fd[0]);        /* read this file     */
      }
      else                     /* parent -> child    */
      {  close(fd[0]);         /* will not be used   */
         return(fd[1]);        /* write this file    */
      }
   }
}
```

The function creates a pipe with the **pipe** function first. It then uses the **fork** function to create two copies of the original process. Each process has its own copy of the pipe. The child process decides whether it is supposed to read or write through the pipe, then closes the other side of the pipe and uses **execl** to create the new process and execute the desired program. The parent, on the other hand, closes the side of the pipe it does not use.

The sequence of **close** functions in the child process links the standard input or output of the child process to the pipe. The first **close** determines which side of the pipe should be closed and closes it. If mode is **WRITE**, the writing side is closed; if **READ**, the reading side is closed. The second **close** closes the standard input or output depending on the mode. If the mode is **WRITE**, the input is closed; if **READ**, the output is closed. The **dup** function creates a duplicate of the side of the pipe still open. Because the standard input or output was closed immediately before this call, this duplicate receives the same file descriptor as the standard file. The system always chooses the lowest available file descriptor for a newly opened file. Because the duplicate pipe has the same file descriptor as the standard file, it becomes the standard input or output file for the process. Finally, the last **close** closes the original pipe, leaving only the duplicate.

The following example is a modified version of the **pclose** function. The modified version requires a file descriptor as an argument rather than a file pointer:

```
#include <signal.h>

pclose(fd)      /* close pipe fd */
int fd;
{
   int r, status;
   int (*hstat)(), (*istat)(), (*qstat)();
   extern int popen_pid;

   close(fd);
   istat = signal(SIGINT, SIG_IGN);
   qstat = signal(SIGQUIT, SIG_IGN);
   hstat = signal(SIGHUP, SIG_IGN);
   while ((r = wait(&status))!= popen_pid && r != -1)
     /* do nothing */ ;
   if (r == -1)
      status = -1;
   signal(SIGINT, istat);
   signal(SIGQUIT, qstat);
   signal(SIGHUP, hstat);
   return(status);
}
```

The function closes the pipe first. It then uses a **while** statement to wait for the child process given by **popen_pid**. If other child processes terminate while it waits, it ignores them and continues to wait for the given process. It stops waiting as soon as the given process terminates or if no child process exists. The function returns the termination status of the child or the value -1 if there was an error.

The **signal** function calls used in this example ensure that no interrupts interfere with the waiting process. The first set of functions causes the process to ignore the interrupt, quit, and hangup signals. The last set restores the signals to their original status. We describe the **signal** function in detail in Chapter 8, "Using Signals."

Both example functions use the external variable **popen_pid** to store the process **ID** of the child process. If you want to open more than one pipe, save the **popen_pid** value in another variable before each call to **popen**. You must restore this value before calling **pclose** to close the pipe. You can modify the functions to support more than one pipe by changing the **popen_pid** variable to an array indexed by a file descriptor.

Chapter 8. Using Signals

This chapter explains how to use C library functions to process signals sent to a program by PC XENIX. A signal is the system's response to an unusual condition that occurs during execution of a program, such as the system detecting an illegal operation or a user pressing the Interrupt key. The Interrupt key is the Del key on your keyboard. A signal interrupts normal execution of the program and initiates an action such as terminating the program or displaying an error message.

The **signal** function of the standard C library lets a program define the action of a signal. The function can disable a signal to prevent it from affecting the program. It can also give a signal a user-defined action.

The **signal** function is often used with the **setjmp** and **longjmp** functions to redefine and reshape the action of a signal. These functions allow programs to save and restore the execution state of a program, giving a program a means to jump from one state of execution to another without a complex assembly language interface.

To use the **signal** function, you must add the line:

```
#include <signal.h>
```

to the beginning of the program. The **signal.h** file defines the various manifest constants used as arguments by the function. To use the **setjmp** and **longjmp** functions you must add the line:

```
#include <setjmp.h>
```

to the beginning of the program. The **setjmp.h** file contains the declaration for the type **jmp_buf**, a template for saving a program's current execution state.

Using the Signal Function

The **signal** function changes the action of a signal from its current action to a given action. The function has the form:

signal(*sigtype*, *ptr*)

where *sigtype* is an integer or a manifest constant that defines the signal you want to change, and *ptr* is a pointer to the function defining the new action or a manifest constant giving a predefined action. The function always returns a pointer value. This pointer defines the signal's previous action and may be used in subsequent calls to restore the signal to its previous value.

The *ptr* may be **SIG_IGN** to indicate no action (ignore the signal) or **SIG_DFL** to indicate the default action. The *sigtype* may be **SIGINT** (for interrupt signal) caused by pressing the Interrupt (Del) key, **SIGQUIT** (for quit signal) caused by pressing the Quit (Ctrl-\) key, or **SIGHANG** (for hangup signal) caused by hanging up the line when connected to the system by modem. Quit is a combination of Ctrl and \. Press and hold down Ctrl and press \. (Constants for other signals are shown under **signal** in *XENIX System Reference.*)

For example, the function call:

```
signal(SIGINT, SIG_IGN);
```

changes the action of the interrupt signal to no action. The signal has no effect on the program. The default action usually terminates the program.

The following sections show how to use the **signal** function to disable, change, and restore signals.

Disabling a Signal

You can disable a signal, that is, prevent it from affecting a program, by using the **SIG_IGN** constant with **signal**. The function call has the form:

signal(*sigtype*, SIG_IGN)

where *sigtype* is the manifest constant of the signal you wish to disable. For example, the function call:

```
signal(SIGINT, SIG_IGN);
```

disables the interrupt signal.

The function call prevents a signal from terminating a program executing in the background (for example, a child process that is not using the terminal for input or output). The system passes signals generated from keystrokes at a terminal to all programs that have been invoked from that terminal. This means that pressing the Interrupt (Del) key to stop a program running in the foreground stops a program running in the background if it has not disabled that signal. For example, in the following program fragment, **signal** disables the interrupt signal for the child:

```
#include <signal.h>

main()
{
    if (fork() == 0)
    {
        signal(SIGINT, SIG_IGN);
        /* Child process. */
    }
    /* Parent process. */
}
```

This call does not affect the parent process, which continues to receive interrupts as before. If the parent process is interrupted, the child process continues to execute until it reaches its normal end.

Restoring a Signal's Default Action

You can restore a signal to its default action by using the **SIG_DFL** constant with **signal**. The function call has the form:

signal(*sigtype*, SIG_DFL)

where *sigtype* defines the signal you wish to restore. For example, the function call:

```
signal(SIGINT, SIG_DFL)
```

restores the interrupt signal to its default action.

The function call restores a signal after it has been temporarily disabled to keep it from interrupting critical operations. For example, in the following program fragment, the second call to **signal** restores the signal to its default action:

```
#include <signal.h>
#include <stdio.h>

main()
{
   FILE *fp;
   char record[512], *filename;

   signal(SIGINT, SIG_IGN);
   fp = fopen(filename, "a");
   fwrite(record, 1, 512, fp);
   signal(SIGINT, SIG_DFL);
}
```

In this example, the interrupt signal is ignored while a record is read from the file given by **fp**.

Catching a Signal

You can catch a signal and define your own action for it by providing a function that defines the new action and giving the function as an argument to **signal**. The function call has the form:

`signal(`*sigtype*`,` *newptr*`)`

where *sigtype* defines the signal you want to catch, and *newptr* is a pointer to the function defining the new action. For example, the function call:

```
signal(SIGINT, catch);
```

changes the action of the interrupt signal to the action defined by **catch**.

The function lets a program do additional processing before terminating. In the following program fragment, **catch** defines the new action for the interrupt signal:

```
#include <stdio.h>
#include <signal.h>

main()
{
   int catch();

   printf("Press Interrupt (Del) key to stop.\n");
   signal(SIGINT, catch);
   while (1)
   {
      /* Body */
   }
}

catch()
{
   printf("Program terminated.\n");
   exit(1);
}
```

Here, **catch** displays the message "Program terminated" before stopping the program with the **exit** function.

A program may redefine the action of a signal at any time. Thus, many programs define different actions for different conditions. For example, in the following program fragment, the action of the interrupt signal depends on the return value of **keytest**:

```
#include <signal.h>

main()
{
   int catch1(), catch2();

   if (keytest() == 1)
      signal(SIGINT, catch1);
      else
         signal(SIGINT, catch2);
}
```

Later, the program may change the signal to the other action or even a third action.

When using a function pointer in the **signal** call, you must make sure that the function name is defined before the call. In the program fragment shown above, **catch1** and **catch2** are explicitly declared at the beginning of the main program function. Their formal definitions are assumed to appear after the **signal** call.

Restoring a Signal

You can restore a signal to its previous value by saving the return value of a **signal** call, then using this value in a subsequent call. The function call has the form:

`signal(`*sigtype*`, `*oldptr*`)`

where *sigtype* is the manifest constant defining the signal you want to restore and *oldptr* is the pointer value returned by a previous **signal** call.

The function restores a signal when its previous action may be one of many possible actions. For example, in the following program fragment, the previous action depends solely on the return value of **keytest**:

```
#include <signal.h>

main()
{
    int catch1(), catch2();
    int (*savesig)();

    if (keytest() == 1)
        savesig = signal(SIGINT, catch1);
        else
            savesig = signal(SIGINT, catch2);
    compute();
    signal(SIGINT, savesig);
}
```

In this example, the old pointer is saved in the variable **savesig**. This value is restored after the function **compute** returns.

Program Example

This section shows how to use the **signal** function to create a modified version of the **system** function. In this version, **system** disables the interrupt and quit signals in the parent process until the child process has completed its operation. It then restores the signals to their previous actions:

```
#include <stdio.h>
#include <signal.h>

system(s)       /* run command string s */
char *s;
{
   int status, pid, w;
   register int (*istat)(), (*qstat)();

   if ((pid = fork()) == 0)
   {
      execl("/bin/sh", "sh", "-c", s, NULL);
      exit(127);
   }
   istat = signal(SIGINT, SIG_IGN);
   qstat = signal(SIGQUIT, SIG_IGN);
   while ((w = wait(&status)) != pid && w != -1)
      /* do nothing */ ;
   if (w == -1)
      status = -1;
   signal(SIGINT, istat);
   signal(SIGQUIT, qstat);
   return(status);
}
```

The parent uses the **while** statement to wait until **wait** returns the child's process identification number (PID). If **wait** returns the error code "-1," no more child processes are left, so the parent returns the error code as its own status.

Controlling Execution with Signals

Signals have uses other than immediately terminating a program. Many signals can be redefined to delay their actions or even cause actions that terminate a portion of a program without terminating the entire program. The following sections describe ways that signals can be caught and used to provide control of a program.

Delaying a Signal's Action

You can delay the action of a signal by catching the signal and redefining its action to be nothing more than setting a globally-defined flag. Such a signal does nothing to the current execution of the program. Instead, the program continues uninterrupted until it can test the flag to see if a signal has been received. It can then respond according to the value of the flag.

The key to a delayed signal is that all functions return execution to the exact point at which the program was interrupted. If the function returns normally, the program continues execution just as if no signal occurred.

Delaying a signal is especially useful in programs that must not be stopped at an arbitrary point. If, for example, a program updates a linked list, the action of a signal is delayed to prevent the signal from interrupting the update and destroying the list. For example, in the following program fragment, the function **delay**, used to catch the interrupt signal, sets the globally-defined flag **sigflag** and returns immediately to the point of interruption:

```
#include <signal.h>

int sigflag = 0;

main()
{
   int delay();
   int (*savesig)();

   savesig = signal(SIGINT, delay); /* Delay the signal. */
   updatelist();
   signal(SIGINT, savesig);
   if (sigflag)
{
    . . .    /* Process delayed signals, if any.      */
   }
}

delay()
{
   sigflag = 1;
}
```

In this example, if the signal is received while **updatelist** is executing, it is delayed until after **updatelist** returns.

The system automatically resets a signal to its default action immediately after the signal is processed. If your program delays a signal, make sure that the signal is redefined after each interrupt. Otherwise, the default action will be taken on the next occurrence of the signal.

Using Delayed Signals with System Functions

When a delayed signal interrupts the execution of a system function, such as **read** or **wait**, the system forces the function to stop and return an error code. This action, unlike actions taken during execution of other functions, causes all processing performed by the system function to be discarded. A serious error can occur if a program interprets a system function error caused by delayed signals as a normal error. For example, if a program receives a signal when reading the terminal, all characters read before the interruption are lost, making it appear as though no characters were typed.

Whenever a program intends to use delayed signals during calls to system functions, the program should include a check of the function return values to ensure that an error was not caused by an interruption. In the following program fragment, the program checks the current value of the interrupt flag **intflag** to make sure that the value **EOF** returned by **getchar** indicates the end of the file:

```
if (getchar() == EOF)
   if (intflag)
      /* EOF caused by interrupt */
      else
         /* true end-of-file     */
```

Using Signals in Interactive Programs

Signals can be used in interactive programs to control the execution of the program's various commands and operations. For example, a signal can be used in a text editor to interrupt the current operation (for example, displaying a file) and return the program to a previous operation (for example, waiting for a command).

To provide this control, the function that redefines the signal's action must be able to return execution of the program to a meaningful location, not just to the point of interruption. The standard C library provides two functions to do this: **setjmp** and **longjmp**. The **setjmp** function saves a copy of a program's execution state. The **longjmp** function changes the current execution state to a previously saved state. The functions cause a program to continue execution at an old location with old register values and status as if no operations had been performed between the time the state was saved and the time it was restored.

The **setjmp** function has the form:

setjmp(*buffer*)

where *buffer* is the variable to receive the execution state. You must declare it with type **jmp_buf** before you use it in the call. For example, in the following program fragment, **setjmp** copies the execution of the program to **oldstate** defined with type **jmp_buf**:

```
jmp_buf oldstate;

setjmp(oldstate);
```

After a **setjmp** call, the *buffer* variable contains values for the program counter, the data and address registers, and the process status. These values must not be modified in any way.

The **longjmp** function has the form:

longjmp(*buffer*)

where *buffer* is the variable containing the execution state. It must contain values previously saved with a **setjmp** function. The function copies the values in the *buffer* variable to the program counter, data and address registers, and the process status table. Execution continues as if it had just returned from the **setjmp** function that saved the previous execution state. For example, in the following program fragment, **setjmp** saves the execution state of the program at the location just before the main processing loop and **longjmp** restores it on an interrupt signal:

```
#include <signal.h>
#include <setjmp.h>

jmp_buf sjbuf;

main()
{
   int onintr();

   setjmp(sjbuf);
   signal(SIGINT, onintr);
   /* main processing loop */
}

onintr()
{
   printf("\nInterrupt\n");
   longjmp(sjbuf);
}
```

In this example, the action of the interrupt signal as defined by **onintr** is to print the message `Interrupt` and restore the old execution state. When an interrupt signal is received in the main processing loop, execution passes to **onintr**, which prints the message, then passes execution back to the main program function, making it appear as though control is returning from the **setjmp** function.

Using Signals in Multiple Processes

PC XENIX passes all signals generated at a given terminal to all programs invoked at that terminal. This means that a program has access to a signal even if that program is executing in the background or as a child to some other program.

Protecting Background Processes

Any program invoked by way of the shell using the shell's background symbol (&) is executed as a background process. Such programs usually do not use the terminal for input or output. Also, they complete their tasks without returning the prompt. Because these programs do not need additional input, the shell automatically disables the signals before executing the program. This means signals generated at the terminal do not affect execution of the program. This is how the shell protects the program from signals intended for other programs invoked from the same terminal.

A program that you have invoked as a background process may try to catch its own signals. If it succeeds, the protection from interruption given to it by the shell is defeated, and signals intended for other programs will interrupt the program. To prevent this, any program you intend to execute as a background process should test the current state of a signal before redefining its action. A program should redefine a signal only if the signal has not been disabled. For example, in the following program fragment, the action of the interrupt signal is changed only if the signal is not currently being ignored:

```
#include <signal.h>

main()
{
   int catch();
   if (signal(SIGINT, SIG_IGN) != SIG_IGN)
      signal(SIGINT, catch);

   /* Program body. */
}
```

This step lets a program continue to ignore signals if it is already doing so and change the signal if it is not.

Protecting Parent Processes

A program can create and wait for a child process that catches its own signals only if the program protects itself by disabling all signals before calling the **wait** function. By disabling the signals, the parent process prevents signals intended for the child processes from terminating its call to **wait**. This prevents serious errors that may result if the parent process continues execution before the child processes are finished.

For example, in the following program fragment, the interrupt signal is disabled in the parent process immediately after the child is created:

```
#include <signal.h>

main()
{
   int status;
   int (*saveintr)();

   if (fork() == 0)
      execl( . . . );

   saveintr = signal(SIGINT, SIG_IGN);
   wait(&status);
   signal(SIGINT, saveintr);
}
```

The signal's action is restored after the **wait** function returns normal control to the parent.

Chapter 9. Using System Resources

Introduction

This chapter discusses how to access the following services of the operating system from a program:

- Message queues
- Semaphores
- Shared memory.

See *XENIX System Reference* for reference information about system calls described in this chapter.

Allocating Space

Some programs require significant changes to the size of their allocated memory space during different phases of their execution. The memory allocation functions of the standard C library let programs allocate space dynamically. This means a program can request a given number of bytes of storage for its exclusive use at the moment it needs the space, then free this space after it has finished using it.

There are four memory allocation functions: **malloc**, **calloc**, **realloc**, and **free**. Use the **malloc** and **calloc** functions to allocate space for the first time. The functions allocate a given number of bytes and return a pointer to the new space. The **realloc** function reallocates an existing space (changes the size of the space), allowing you to use it in a different way. The **free** function returns allocated space to the system.

Allocating Space for a Variable

The **malloc** function allocates space for a variable containing a given number of bytes. The function call has the form:

```
malloc(size)
```

where *size* is an unsigned number that gives the number of bytes you want to allocate. For example, the function call:

```
table = malloc(4);
```

allocates 4 bytes of storage. The function normally returns a pointer to the starting address of the allocated space. It returns the null pointer value if there is not enough space to allocate.

The function allocates storage for a group of strings that vary in length. For example, in the following program fragment, **malloc** allocates space for 10 strings, each of different length:

```
char * malloc();
int i;
char *temp, *strings[10];
unsigned isize;

for (i = 0; i < 10; i++)
{
   scanf("%s", temp);
   isize = strlen(temp) + 1;
   strings[i] = malloc(isize);
}
```

In this example, the strings are read from the standard input. The **strlen** function gets the size in bytes of each string.

Allocating Space for an Array

The **calloc** function allocates storage for a given array and initializes each element in the new array to zero. The function call has the form:

```
calloc(n, size)
```

where *n* is the number of elements in the array, and *size* is the number of bytes in each element. The function normally returns a pointer to the starting address of the allocated space. It returns a null pointer value if there is not enough memory. For example, the function call:

```
table = calloc(10, 4);
```

allocates sufficient space for a 10-element array. Each element occupies 4 bytes.

The *calloc* function is used in programs that must process large arrays without knowing the size of an array in advance. For example, in the following program fragment, **calloc** allocates storage for an array of values read from the standard input:

```
char *calloc();
int  element_count;
int  *table;
int  i;

scanf("%d", &element_count);
table = (int *) calloc(element_count, sizeof(int));
for (i = 0; i < element_count; i++)
   scanf("%d", table[i]);
```

The number of elements, **inum**, is read from the standard input before the elements are read.

Reallocating Space

The **realloc** function reallocates the space at a given address without changing the contents of the memory space. The function call has the form:

`realloc(`*ptr*`,` *size*`)`

where *ptr* is a pointer to the starting address of the space you want to reallocate, and *size* is an unsigned number giving the new size in bytes of the reallocated space. The function normally returns a pointer to the starting address of the allocated space. It returns a null pointer value if there is not enough space to reallocate the storage.

This function keeps storage as compact as possible. For example, in the following program, **realloc** reads a variable-sized array of integers from standard input and process the array using the function `do_something`:

```
main()
{
    char *realloc(), *malloc();
    int  *array_ptr;
    int  number_of_arrays;
    int  array_size;
    int  i, j;

    /* Give the pointer an initial value. */
    array_ptr = (int *) malloc(sizeof(int));
    /* . . . */
    scanf("%d", &number_of_arrays);

    for (i = 0; i < number_of_arrays; i++)
    {
        scanf("%d", &array_size);
        /* resize the data area */
        array_ptr = realloc(array_ptr, array_size * sizeof(int)
        /* read in the array elements */
        for (j = 0; j < array_size; j++)
        {
            scanf("%d", array_ptr[j]);
        }
        printf("Result is: %d\n",
        do_something(array_ptr, array_size));
    }
    free(array_ptr);
}
```

Freeing Unused Space

The **free** function frees unused memory space that had been previously allocated by a **malloc**, **calloc**, or **realloc** function call. The function call has the form:

`free(`*ptr*`)`

where *ptr* is the pointer to the starting address of the space you want to free. This pointer must be the return value of a **malloc**, **calloc**, or **realloc** function.

The function frees space that is no longer used or frees space you can use for other purposes. For example, the following program fragment frees some previously allocated (using **malloc**) storage:

```
main()
{
   char *malloc(), *p;
   p = malloc(SOME_SIZE);
   /* perform operations on p */
   free(p); /* give back the storage */
}
```

Locking Files

Locking a file is a way to synchronize file use when several processes may require access to a single file. The standard C library provides one file locking function, **lockf**. This function locks any given section of a file, preventing all other processes that wish to use the section from gaining access. A process may lock the entire file or only a small portion of it. In any case, only the locked section is protected; all other sections can be accessed by other processes as usual.

File locking protects a file from the damage caused by several processes trying to read or write to the file at the same time. File locking also provides unhindered access to any portion of a file for a controlling process. Before a file can be locked, however, it must be prepared using the **open** and **lseek** functions described in Chapter 3, "Using the Standard I/O Functions." To use the **lockf** function, you must add the line:

```
#include <unistd.h>
```

to the beginning of the program. The file **unistd.h** contains definitions for the modes used with the function.

Preparing a File for Locking

Before you can lock a file, you must first open it using the **open** function, then properly position it with the **lseek** function to move the file's character pointer to the first byte you want to lock.

Use the **open** function once at the beginning of the program to open the file. You can use the **lseek** function any number of times to move the character pointer to each new section you want to lock. For example, the following statements locate the file pointer at the byte position 1024 from the beginning of the file **reservations** for locking:

```
fd = open("reservations",O_RDONLY);
lseek(fd, 1024, 0);
```

Locking a File

The **lockf** function locks 1 or more bytes of a given file. The function call has the form:

`lockf(`*fildes*`, `*function*`, `*size*`)`

where *fildes* is the file descriptor of the file locked, *function* is an integer value that defines the type of lock applied to the file, and *size* is a long integer value giving the size in bytes of the portion of the file section locked or unlocked. The *function* may be **F_LOCK** for locking the given bytes or **F_ULOCK** for unlocking them. For example, in the following program fragment, **lockf** locks 100 bytes at the current character pointer position in the file given by **fd**:

```
#include <unistd.h>

main()
{
   int fd;

   fd = open("data", O_RDWR);
   lockf(fd, F_LOCK, 100L);
}
```

The function normally returns **0** (zero), but returns -1 if it encounters an error.

Program Example

This section shows how to lock and unlock a small section in a file using the **lockf** function. Do an **lseek** to a specified position immediately before the locking or unlocking of the file segment. It is advisable to do this because the **lockf** call takes the current file pointer position as the starting point of the segment you want to lock. In the following program, the file **data** opens for reading and writing, **lseek** sets the file pointer correctly, and then the **lockf** function locks 100 bytes of the file. A **lseek** again sets the file pointer to the specified location and then the **lockf** call unlocks the the file.

```
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>

main()
{
   int fd, err;
   char *data;

   fd = open("data", O_RDWR);            /* Open data for R/W     */
   if (fd == -1)
      perror("");
   else
   {
      lseek(fd, 100L, 0);                /* Seek to pos 100       */
      err = lockf(fd, F_LOCK, 100L);     /* Lock bytes 100-200    */
      if (err == -1)
      {
          /* process error return */
      }
      /* read or write bytes 100 - 200 in the file */
     lseek(fd, 100L, 0);                 /* Seek to pos 100       */
     lockf(fd, F_ULOCK, 100L);           /* Unlock bytes 100-200 */
   }
}
```

Header Files Needed for Calls

Some system calls depend on special macro definitions and declarations for the values that they return. The system provides this information in files called ***header files*** that are in the directory **/usr/include**. Therefore, when using system calls, ensure to also include any header files that the system call needs. To include a file in a program, use the following statement in the program:

```
#include <file.h>
```

where the parameter, *file.h*, represents the name of the header file to use.

The header files needed for each system call are:

Header File	**Calls That Use The Header File**
sys/types.h	msgop( ), msgctl( ), msgget( ), semctl( ), semget( ), semop( ), shmctl( ), shmget( ), shmop( )
sys/ipc.h	msgop( ), msgctl( ), msgget( ), semctl( ), semget( ), semop( ), shmctl( ), shmget( ), shmop( )
sys/msg.h	msgop( ), msgctl( ), msgget( ),
sys/shm.h	shmctl( ), shmget( ), shmop( )
sys/sem.h	semctl( ), semget( ), semop( )

Note: When compiling a program that contains any of the **#include** files **msg.h**, **sem.h**, or **shm.h**, you must compile the program with the **-Me** option.

Semaphore Calls

Semaphores provide a general method of communication between two processes that is an extension of the features of signals. Use semaphores in much the same way as signals, except that semaphores are:

More flexible: Processes can define a semaphore to mean what they want it to mean.

More controllable: Programs have direct control over semaphores and do not need to depend on the system to generate them.

Broader in scope: A semaphore can be any integer value, not just 1 or 0. Use them for counting as well as process coordination (see program example).

The system calls that allow a program to use semaphores are:

Call	**Description**
semctl( )	Semaphore control operations
semget( )	Get set of semaphores
semop( )	Semaphore operations

Note: When compiling a program that contains the **#include** file **sem.h**, you must compile the program with the **-Me** option.

Structure of a Semaphore Set

When using the **semget( )** system call to create a set of semaphores, the system returns an integer that is the ***semid***, or semaphore ID, for the set of semaphores that were created. Each semid points to a set of semaphores and a data structure that contains information about the semaphores. The data structure for semid is shown in Figure 9-1 on page 9-9. See Figure 9-2 on page 9-9 for the data structure of a semaphore.

Name	Function
sem_perm	Operation Permission Struct
cuid	creator user ID
cgid	creator group ID
uid	user ID
gid	group ID
mode	read and alter permission
sem_nsems	Number of Semaphores in the Set
sem_otime	Time of the Last Operation (seconds since 1/1/1970)
sem_ctime	Time of the Last Change (seconds since 1/1/1970)

Figure 9-1. Semid Data Structure

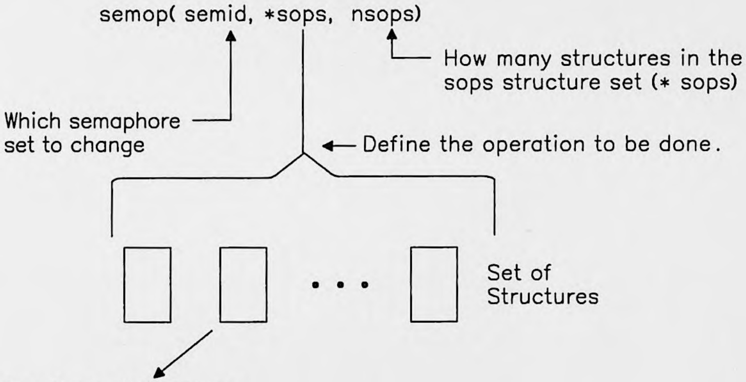
Name	Function
semval	Value of the semaphore (0 or positive)
sempid	Process ID of the Last Operation
semncnt	The number of processes that are waiting for semval to be > current value of its last semop( ) call
semzcnt	The number of processes that are waiting for semval to be = 0.

Figure 9-2. Semaphore Structure

How to Use Semaphores

Semaphores are counters that a program can test and change with a single system call (**semop( )**). Specify the amount of the change in the **semop( )** call using the **sem_op** variable within the **sops** structure as shown in Figure 9-3 on page 9-10. When using this call, the system tests the value of **sem_op** against the value of the semaphore indicated by **sem_num**. If (sem_flg & IPC_NOWAIT) is true, the call returns without further action. If it is false, the table in Figure 9-4 on page 9-11 summarizes the actions that occur.

Use semaphores for passing data between processes and for other one time data transfers. You can also use them to control access to a limited resource, such as a shared buffer.


semop(semid, *sops, nsops)
How many structures in the sops structure set (* sops)
Which semaphore set to change
Define the operation to be done.
Set of Structures

One Typical Structure:

Name	Function
sem_num	Number of the semaphore to be changed
sem_op	The semaphore operation
sem_flg	Operation flags

Figure 9-3. semop( ) System Call Parameters

Value of sem_op	Relationship to semval	Actions
< 0	|sem_op| <= semval	1. semval = semval - | sem_op |
	|sem_op| > semval	1. semncnt = semncnt + 1 2. wait for semval >=| sem_op |; then semval = semval - | sem_op | semncnt = semncnt - 1
> 0		1. semval = semval + sem_op
0	semval== 0	1. return
	semval !=0	1. semzcnt = semzcnt + 1 2. wait for semval = 0; then semzcnt = semzcnt - 1 return

Figure 9-4. How sem_op Specifies a Semaphore Operation

Example of Semaphores

For example, two processes `proca` and `procb` share a buffer `bufr`. `proca` produces data packages and places each package in the buffer as it produces that package. `procb` uses the data packages from the buffer in its operation, but does not use them at the same rate that `proca` puts them in the buffer. The buffer can hold only three data packages. The processes use two semaphores to ensure that:

- `proca` does not try to put a package in a full buffer.
- `procb` does not try to take a package from an empty buffer.

The processes agree that the semaphores have the following meanings:

sem1	The number of empty slots in the buffer. If sem1 is greater than zero, then `proca` can put a package in the buffer.
sem2	The number of data packages in the buffer. If sem2 is greater than zero, then `procb` can take a package from the buffer.

Figure 9-5 on page 9-12 shows the relation of the processes to the buffer and semphores. When the processes start, `sem1` has a value of 3 (empty slots) and `sem2` has a value of 0 (packages in `bufr`).

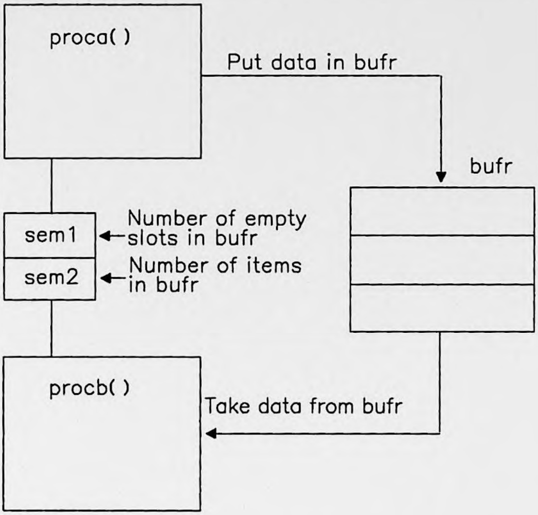

proca( )
Put data in bufr
bufr
sem1
Number of empty slots in bufr
sem2
Number of items in bufr
procb( )
Take data from bufr

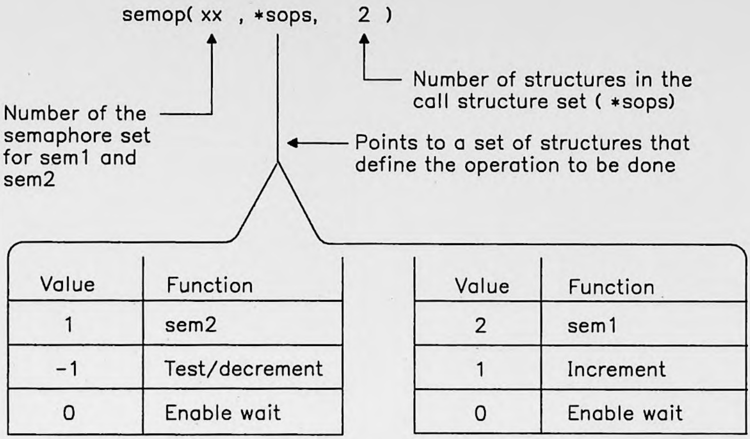
Figure 9-5. Using Semaphores Concept Example

proca

`proca` puts items into the buffer. Before it can put an item in the buffer, it tests `sem1` to find out if there is room in the buffer for the item. If `sem1` is greater than, or equal to 1 (the number of items to be put into `bufr`), then it:

1. Decrements `sem1` by 1
2. Puts an item in the buffer
3. Increments `sem2` by 1.

If `sem1` is less than 1, `proca` waits until there is room in the buffer (`sem1` > = 1), and then performs the preceding steps. If `proca` has to wait, it increments the `semncnt` flag for `sem1` to indicate that it is waiting. It decrements this flag when it continues. Figure 9-6 on page 9-13 shows the **semop( )** call and structure that handles all of the semaphore operations for `proca`.


semop(xx , *sops, 2)
Number of the semaphore set for sem1 and sem2
Points to a set of structures that define the operation to be done
Number of structures in the call structure set (*sops)

Value	Function
1	sem2
-1	Test/decrement
0	Enable wait

Value	Function
2	sem1
1	Increment
0	Enable wait

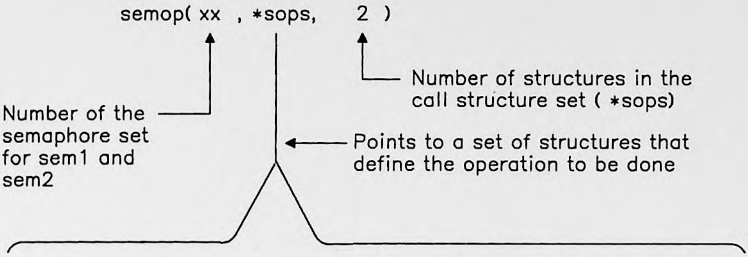
Figure 9-6. semop( ) Call for Proca

procb

`procb` takes items from the buffer. Before it can take an item, it tests `sem2` to find out if there is anything in the buffer. If `sem2` is greater than, or equal to 1 (the number of items to be taken), then it:

1. Decrements `sem2` by 1
2. Takes an item from the buffer
3. Increments `sem1` by 1.

If `sem2` is less than 1, `procb` waits until there is something in the buffer (`sem2` > = 1), and then performs the preceding steps. If `procb` has to wait, it increments the `semncnt` flag for `sem2` to indicate that it is waiting. It decrements this flag when it continues. Figure 9-7 on page 9-14 shows the **semop( )** call and structure that handles all of the semaphore operations for `procb`.


semop(xx , *sops, 2)
Number of the semaphore set for sem1 and sem2
Number of structures in the call structure set (*sops)
Points to a set of structures that define the operation to be done

Value	Function
2	sem2
-1	Test/decrement
0	Enable wait

Value	Function
1	sem1
1	Increment
0	Enable wait

Figure 9-7. semop( ) Call for procb

Operation

The chart in Figure 9-8 on page 9-15 shows the operation of the semaphores to control access to the buffer.

Event	sem1		sem2	
	Value	semncnt	Value	semncnt
1.Start: The initial state of the parameters	3	0	0	0
2.Procb tries to get item:	3	0	0	1
3.Proca puts item in bufr.	2	0	1	1
4.Procb can now get item.	3	0	0	0
5.Proca puts item in bufr.	2	0	1	0
6.Proca puts item in bufr.	1	0	2	0
7.Proca puts item in bufr.	0	0	3	0
8.Proca tries to put item in bufr.	0	1	3	0
9.Procb gets item from bufr.	1	1	2	0
10.Proca can now put item in bufr.	0	0	3	0

Figure 9-8. Semaphore Usage

Example of Semaphore Programming

The program in Figure 9-9 on page 9-16 shows the use of semaphores in a situation similar to the example concept described previously. This program also uses some shared memory calls as described in "Shared Memory Calls" on page 9-25. You can compile and run this program to see the effects of the system calls using the following command (source code for the program is in a file that you create, called **semtst.c**):

```
cc -Mes -i semtst.c
```

```
/* This program is an example producer/consumer process pair
** that shows the use of semaphores to regulate access to a
** buffer.  It also illustrates the use of shared memory
** because the buffer is set up in a shared memory segment.
** The program creates two processes that cooperate in
** production and consumption of an 'item'.  Both processes
** are descendants of the same parent process.  They do not
** have to be, however, when using semaphores in a more
** general application.
*/

#define STKSZ          2    /* size of buffer              */
#define SNSLTS         0    /* number of empty slots       */
#define SNITMS         1    /* number of items in buffer   */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <errno.h>
#define MADDR          (char far *) 0
                            /* Sets shmflg to 0 to indicate */
                            /* that the segment is attached */
                            /* to the 1st available address */

int semid;
int shmid;
extern int errno;
struct ITM                  /* shared memory template       */
  {
     char *itmstk[STKSZ];   /* item buffer stack array      */
     int sp;                /* stack pointer for buffer     */
  }  far *stkptr;

main()
{
  char *popitm();           /* routine to get item from buf */
  int nsems;                /* number of sems/sem flag      */
  int pid_1, pid_2;          /* process id of two processes  */
  key_t key, ftok();        /* key to get shared memory     */
  int shmdt();              /* shared memory detach call    */
  struct sembuf soa1[1], soa2[1]; /* semaphore buffer       */
  struct sembuf *sops1 = soa1,
                *sops2 =  soa2; /* pointer to buffer        */
```

Figure 9-9 (Part 1 of 4). Using Semaphore Calls

```
/* The following section gets a key, sets the number of
** semaphores to 2, and gets the semid and data structure
** associated with the key.
** NOTE: The finished program does not work if the file
** "semtst.c" is not in the current directory when the
** program runs.  Then the program initializes semaphore 0
** to a value of 2, and semaphore 1 to a value of 0.
*/

key = ftok("semtst.c", 's');
nsems = 2;

semid = semget(key, nsems, IPC_CREAT ¦ SEM_A ¦ SEM_R);
semctl(semid, SNSLTS, SETVAL, STKSZ);
semctl(semid, SNITMS, SETVAL, 0);

/*
** The next section gets a shared memory segment and its
** associated id, using the same key value and a memory
** segment size of STKSZ.  Then it attaches the shared
** memory segment to the data segment of the process (still
** one process).
*/

shmid = shmget(key, sizeof(*stkptr), IPC_CREAT ¦ 0666);
stkptr = ( (struct ITM far *) shmat(shmid, MADDR,
                                IPC_CREAT ¦ 0666) );

/* The first fork sets up the producer process.  It
** initializes the stack pointer (for the buffer) to zero
** and begins producing items to put into the buffer.
** When the buffer is full (empty slots = 0) the process
** waits for an empty slot to appear.
*/

if( (pid_1 = fork() ) == 0)
{
   int i;

   stkptr->sp = 0;
   for(i = 0; i <= 5; i++)
   {
      printf( "\nProducer: Sending item number %d   ", i );
      p(SNSLTS, sops1);   /* decrement no of empty slots */
      pushitm("Test Item", stkptr); /* put item in shared area */
      v(SNITMS, sops2);   /* increment no of items       */
   }
   exit(0);
}   /* end producer */
```

Figure 9-9 (Part 2 of 4). Using Semaphore Calls

```
  /* The second fork sets up the consumer process. It begins
  ** taking items from the buffer.  When the buffer is empty
  ** (empty slots = 2) the process waits for another item to
  ** appear.
  */

  if( (pid_2 = fork() ) == 0)
  {
     int i;
     char *itmptr;

     for(i = 0;i <= 5;i++)
     {
        p(SNITMS, sops1);          /* decrement no of items     */
        semctl( semid, 1, GETVAL);
        itmptr = popitm(stkptr);   /* pop item from shared area */
        v(SNSLTS, sops2);        /* increment no of empty slots */
        printf("\nConsumer: Got item number %d; item is '%s'",
               i, itmptr);
     }
     exit(0);
  }
  /* Parent process waiting for death of child processes. */
  wait( (int *) 0);
  wait( (int *) 0);
}   /* End Parent Process */

/* The following section implements p(s) and v(s), the wait/signal
** routines for the producer and consumer.  Producer can produce
** if the number of empty slots is not 0.  Consumer can consume if
** the number of items is not 0.
*/

int stpop, stvop;

p(s, sops)
int s;
struct sembuf *sops;
{
  int nsops;

  nsops = 1;                      /* no of semaphore operations = 1 */
  sops->sem_num = s;                 /* assign semaphore number     */
  sops->sem_flg = ~IPC_NOWAIT;        /* IPC_NOWAIT false           */
  sops->sem_op = -1;                 /* decrement operation         */
  stpop = semop(semid, sops, nsops); /* perform semaphore operation */
  return;
}  /* end p() */
```

Figure 9-9 (Part 3 of 4). Using Semaphore Calls

```
v(s, sops)
int s;
struct sembuf *sops;
{
  int nsops;

  nsops = 1;                      /* no of semaphore operations = 1 */
  sops->sem_num = s;                 /* assign semaphore number     */
  sops->sem_flg = ~IPC_NOWAIT;       /*  IPC_NOWAIT                 */
  sops->sem_op = 1;                  /* increment operation         */
  stvop = semop(semid, sops, nsops); /* perform semaphore operation */
  return;
} /* end v() */

/* popitm: fetch next item, semphore example */

char *popitm(ip)
struct ITM far *ip;
{
  if(ip->sp > 0)
  {
     printf("\npop: stack pointer = %d   ", --ip->sp);
                                 /* decrement stack pointer   */
     return(ip->itmstk[ip->sp]); /* return pointer to message */
  }
 else
     return(NULL);
} /* end popitm */

/* pushitm: push item onto buffer/stack */

int s;                                    /* semaphore number */
pushitm(item, ip)
char *item;
struct ITM far *ip;
{
  ip->itmstk[ip->sp++] = item;
           /* push item on stack and increment stack pointer */
  printf("\npush: stack pointer = %d   ", ip->sp);
} /* end pushitm */
```

Figure 9-9 (Part 4 of 4). Using Semaphore Calls

Message Calls

Messages provide a general method of communication between two processes. Using messages one process can pass information of any kind to another process. The information may be data that is produced by one process and used in another, or it could be flags that indicate when events occur. To use the message process, perform the following steps:

1. Use the **ftok( )** subroutine to get a ***key*** assigned to a message queue.
2. Use the **msgget( )** call to get a message queue assigned to the processes.

3. Use the **msgsnd( )** call to send a message to a queue that is assigned to another process.
4. Use the **msgrcv( )** call to receive a message from the message queue.

Use the following system calls to create and use message queues:

Call	**Description**
msgctl( )	Gets status, changes permissions, or removes a queue.
msgget( )	Gets message queue.
msgrcv( )	Receives a message.
msgsnd( )	Sends a message.

In addition, the **ftok( )** subroutine provides the ***key*** that the **msgget( )** call uses to create the message queue. Include the following header files when using message queue calls:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

Note: When compiling a program that contains the **#include** file **msg.h**, you must compile the program with the **-Me** option.

Terms

The use of message queue elements is similar in structure to the way the system creates and uses files. Defining the terms used for message queues with respect to the more familiar file terms provides a framework to build an understanding of message queues. Figure 9-10 shows the new terms and how they relate to terms used with files.

Term	**Definition**
key	The ***key*** is a unique identifier (of type **key_t**) that names the particular message queue. It is always associated with the message queue as long as the message queue exists. In this respect, it is similar to the *filename* of a file.

Figure 9-10 (Part 1 of 2). Message Queue Terms

Term	Definition
msqid	The ***msqid*** is an identifier assigned to the message queue for use within a particular process. It is similar in use to a *file descriptor* of a file.
Permissions	The message queue structure also contains information that describes the access permissions for the message queue. These permissions are similar in function to the access permission bits for a file (owner, group and others).

Figure 9-10 (Part 2 of 2). Message Queue Terms

In effect, message queues are a more general form of the **pipe** system call. Either method passes information between two processes. For message queues, however, you do not need to perform the steps of opening a pipe, forking, and then closing two of the ends of the pipe as you would with a pipe. In fact, the two processes using message queues to communicate do not need to be created from the same ancestor process; they only need to cooperate by using the same name for the queue, and agreeing about what the messages mean.

General Operation

When a process gets a message queue, it uses an internal name (***key***) to apply to the queue. Any other programs that use that key can access that queue, subject to the read/write access permissions set up for the queue. The process can either send messages to the queue or receive messages from the queue (or both if it wants to send messages to itself). If a process tries to get a message from the queue and no messages are in the queue, the process is blocked until a message is received. To avoid having the process blocked, set the `IPC_NOWAIT` option in the queue request.

After opening the queue, the process continues operating until it reaches the point that it needs input from the other cooperating process. The first process checks its message queue using the **msgrcv( )** call. Using the parameter, ***msgtype***, in the **msgrcv( )** call, the process can specify which type of message it wants to receive. If a message that satisfies the request is not in the queue, the first process halts until something is put into the queue that does satisfy the request. If there is a message of the requested type in the queue, the system gives the process the first message of that type that was put into the queue (first-in-first-out).

Similarly, after opening the queue, the other process can send messages to the queue of the first process. If the queue is full, the system returns an error indication and the process must wait until the first process empties the queue enough to add the new message.

Because either of these wait conditions could halt the process indefinitely, the program should include a timeout loop to end the stalled condition.

Sending messages to a queue is completely independent from receiving messages from that queue. The amount of data that one process can put into the message queue of the other process depends on the queue size and the speed that the other process takes the data from the queue. More than one process can put messages into a queue. The receiving process must take them out of the queue in the order that they were put into the queue, modified only by selecting a message type.

Using Message Queues

The following sequence shows how to create and use a message queue:

1. Create a key to uniquely identify the message queue. Use the **ftok( )** subroutine to create the key. For example, to create a key `mykey` using a project ID of `x` contained in the **char** variable `proj` and a file name of `null_file`, use a statement like:

   ```
   mykey = ftok( null_file, proj );
   ```

2. Either:

   - Create a new message queue with the **msgget( )** system call. For example, to create a message queue and assign the **msqid** to an integer variable `msg_qid`, use a statement like:

     ```
     msg_qid = msgget( mykey, IPC_CREAT );
     ```

     or

   - Get a previously created message queue with the **msgget( )** system call. For example, to get a message queue that is already associated with the key `mykey` and assign the **msqid** to an integer variable `msg_qid`, use a statement like:

     ```
     msg_qid = msgget( mykey, IPC_ACCESS );
     ```

3. Use the queue to send or receive messages with other processes.
4. If the queue is no longer needed, eliminate it from the system using the **msgctl( )** system call:

```
msgctl( msg_qid, IPC_RMID );
```

See *XENIX System Reference* for specific information about parameters for the calls and subroutines.

Example of Message Queue Calls

The program in Figure 9-11 shows the use of message queues in a simple producer-consumer relationship. One process produces an item for the other process and passes it to the other process on a message queue. You can compile and run this program to see the effects of the system calls using the following command (source code for the example program is in a file that you create, called **msgqtst.c**):

```
cc -Mes -i msgqtst.c
```

```
/*  msgqtst.c: consumer/producer illustrating message queues.
**  Creates two processes that cooperate in production and
**  consumption of an 'item'
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>
#define MSGSIZ          15  /* common size                  */
#define MSGRTYP          0  /* receive first message on queue */
#define MTYPE            1  /* common message type          */

static char *msg[11] = {
                         "this is item 0", "this is item 1",
                         "this is item 2", "this is item 3",
                         "this is item 4", "this is item 5",
                         "this is item 6", "this is item 7",
                         "this is item 8", "this is item 9",
                         "this is item 10"
                       };
int  msgqid;
extern int errno;
```

Figure 9-11 (Part 1 of 3). Example of Using Message Queues

```
main()
{
  int pid_1, pid_2;
  key_t key, ftok();
  char *receive();
  int send();

/*  The following code creates a key using the path name for the
**  source file and a project id of the letter 'm'.  If you run
**  this program in a directory that does not contain the file
**  "msgqtst.c", the program will not work.  The code then gets a
**  message queue ID, a message queue, and a data structure, and
**  forks into two child processes.  The parent process waits for
**  the children to die.
*/

  key = ftok("msgqtst.c", 'm');
  msgqid = msgget(key, IPC_CREAT | MSG_R | MSG_W);
  printf("msgqid = %d\n", msgqid);
  if( (pid_1 = fork() ) == 0)
  {
                   /* Begin Child-1 Process */
     int i ;

     for(i = 0; i <= 10; i++)
     {
        printf("Producer process: sending message number %d\n", i);

        send(msg[i]); /* produce item */
     }
     exit(0);
  }   /* End Child-1 Process   */
  if( (pid_2 = fork() ) == 0)
  {   /* Begin Child-2 Process */
     int i;

     for(i = 0; i <= 10; i++)
     {
         printf("Consumer process: message number %d is: %s\n",
                   i, receive() ); /* consume item */
     }
     exit(0);
  }   /* End Child-2 Process */
  /* Parent Process waiting for death of child processes.  */
  wait( (int *) 0);
  wait( (int *) 0);
}   /* End Parent Process */
```

Figure 9-11 (Part 2 of 3). Example of Using Message Queues

```
/*  The following routine is the send() routine for the
**  producer process.
*/

struct msgbuf msgb;
struct msgbuf *msgp = &msgb; /* define pointer to message buffer */
char *strcpy();

send(item)
char *item;
{
  /* The following code sets the message type to MTYPE (1),
  ** copies the string "item" into the message buffer,
  ** and sends the message.
  */
  msgp->mtype = MTYPE;
  strcpy( (msgp->mtext), item);
  msgsnd(msgqid, msgp, MSGSIZ, ~IPC_NOWAIT);
  return;
}

/*  The following routine is the receive() routine for the
**  consumer process.  It receives the message and returns
**  a pointer to the message.
*/

char *receive()
{
  msgrcv(msgqid, msgp, MSGSIZ, MSGRTYP, ~IPC_NOWAIT);
  return( (msgp->mtext) );
}
```

Figure 9-11 (Part 3 of 3). Example of Using Message Queues

Shared Memory Calls

The ***shared memory*** calls set aside an area of memory that cooperating processes can access. This area can serve as a large pool for exchanging data among the processes. The shared memory calls do not provide locks or access control among the processes. Therefore, processes using the shared memory area must set up a signal or semaphore control method to prevent access conflicts and to keep one process from changing data that another process is using. Use shared memory when the amount of data to be exchanged between processes is too large to transfer with messages, or when many processes maintain a common large data base.

Use the following calls to create and use shared memory segments from a program:

Call	**Description**
shmctl( )	Controls shared memory operations.
shmget( )	Gets or creates a shared memory segment.
shmat( )	Attaches a shared memory segment to a process.
shmdt( )	Detaches a shared memory segment from a process.

In addition, the **ftok( )** subroutine provides the ***key*** that the **shmget( )** call uses to create the shared segment. Include the following header files when using shared memory calls:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

Note: When compiling a program that contains the **#include** file **shm.h**, you must compile the program with the **-Me** option.

Terms

The use of shared memory segments is similar in structure to the way the system creates and uses files. Defining the terms used for shared memory with respect to the more familiar file terms provides a framework to build an understanding of shared memory. Figure 9-12 shows the new terms and how they relate to terms used with files.

Term	**Definition**
key	The ***key*** is a unique identifier that names the particular shared segment. It is always associated with the shared segment as long as the shared segment exists. In this respect it is similar to the *filename* of a file.

Figure 9-12 (Part 1 of 2). Shared Memory Terms

Term	Definition
shmid	The ***shmid*** is an identifier assigned to the shared segment for use within a particular process. It is similar in use to a *file descriptor* of a file.
attach	A process must ***attach*** a shared segment to use the shared segment. Attaching a shared segment is similar to opening a file.
detach	A process must ***detach*** a shared segment once it is finished with the shared segment. Detaching a shared segment is similar to closing a file.

Figure 9-12 (Part 2 of 2). Shared Memory Terms

Using Shared Segments

The following sequence describes the life cycle of a shared segment from initial creation to final removal from the system:

1. Create a key to uniquely identify the shared segment. Use the **ftok( )** subroutine to create the key. For example, to create a key `mykey` using a project ID of `R` contained in the variable `proj` (type **char**) and a file name of `null_file`, use a statement like:

   ```
   mykey = ftok( null_file, proj );
   ```

2. Either:

   - Create the shared memory segment with the **shmget( )** system call. For example, to create a shared segment that contains 4096 bytes and assign the **shmid** to an integer variable `mem_id`, use a statement like:

     ```
     mem_id = shmget( mykey, 4096, IPC_CREAT | 0666 );
     ```

     or

   - Get a previously created shared segment with the **shmget( )** system call. For example, to get a shared segment that is already associated with the key `mykey` and assign the **shmid** to an integer variable `mem_id`, use a statement like:

     ```
     mem_id = shmget( mykey, 4096, IPC_ACCESS );
     ```

3. Attach the shared segment to the process with the **shmat( )** system call. For example, to attach the previously created segment, use a statement like:

   ```
   ptr = shmat( mem_id );
   ```

   In this example, the variable `ptr` is a pointer to a structure that defines the fields in the shared segment. Use this template structure to store and retrieve data in the shared segment. This template should be the same for all processes using the segment.

4. Work with the data in the segment using the template structure.

5. Detach from the segment using the **shmdt( )** system call:

   ```
   shmdt( mem_id );
   ```

6. If the shared segment is no longer needed, remove it from the system using the **shmctl( )** system call:

   ```
   shmctl( mem_id, IPC_RMID, ptr ) ;
   ```

See *XENIX System Reference* for specific information about parameters for the calls and subroutines. You can also use the commands **ipcs** to get information about a segment and **ipcrm** to remove a segment. See *XENIX Commands Reference* for information about these commands.

Chapter 10. Error Processing

This chapter explains how to process errors and describes the functions and variables a program uses to respond to errors.

PC XENIX detects and reports errors that occur when using standard C library functions. Errors range from problems with accessing files to allocating memory. In most cases, the system simply reports the error and lets the program decide how to respond. PC XENIX terminates a program only if a serious error occurs, such as an attempt to access unavailable or protected memory that results in a violation of memory space.

Using the Standard Error File

The standard error file is a special output file to display error messages. This file is one of three standard files (standard input, standard output, and standard error) automatically created for the program when it is invoked.

The standard error file, like the standard output, is normally assigned to the user's terminal screen. Thus, error messages written to the file are displayed at the screen. The file can also be redirected by using the shell's redirection symbol (>). For example, the following command redirects the standard error file to the file **errorlist**:

```
dial  2 > errorlist
```

In this case, subsequent error messages are written to the given file.

The standard error file, like the standard input file and standard output file, has predefined file pointer and file descriptor values. In stream functions, the file pointer **stderr** copies data to the error file. In low-level functions, the file descriptor **2** copies data to the file. For example, in the following program fragment, **stderr** writes the message "Unexpected end of file." to the standard error file:

```
if ((c=getchar()) == EOF)
   fprintf(stderr,"Unexpected end of file.\n");
```

The standard error file is not affected by the shell's pipe symbol (|). This means that even if the standard output of a program is piped to another program, errors generated by the program still appear at the terminal screen (or in the appropriate file if the standard error is redirected).

Using the errno Variable

The **errno** variable is a predefined external variable that contains the error number of the most recent system function error. Errors detected by system functions, such as access permission errors and lack of space, cause the system to set the **errno** variable to a number and return control to the program. The error number identifies the error condition. The variable can be used in subsequent statements to process the error.

The **errno** variable is used immediately after a system function has returned an error. In the following program fragment, **errno** determines the course of action after an unsuccessful call to the **open** function:

```
if ((fd = open("accounts", O_RDONLY)) == -1)
   switch (errno)
   {
      case(EACCES):
         fd = open("/usr/tmp/accounts", O_RDONLY);
         break;
      default:
         exit(errno);
   }
```

In this example, if **errno** is equal to **EACCES** (a manifest constant defined in **errno.h**), permission to open the file **accounts** in the current directory is denied, so the file is opened in the directory **/usr/tmp** instead. If the variable is any other value, the program ends.

To use the **errno** variable in a program, you must explicitly define it as an external variable with int type. The file **errno.h** contains manifest constant definitions for each error number. These constants can be used in any program in which the line:

```
#include <errno.h>
```

appears at the beginning of the program.

Printing Error Messages

The **perror** function copies a short error message describing the most recent system function error to the standard error file. The function call has the form:

```
perror(s)
```

where *s* is a pointer to a string containing additional information about the error.

The **perror** function places the given string before the error message and separates the two with a colon. Each error message corresponds to the current value of the **errno** variable. For example, in the following program fragment, **perror** displays the message "accounts: Permission denied" if **errno** is equal to the constant **EACCES**:

```
if (errno == EACCES)
{
   perror("accounts");
   fd = open (/usr/tmp/accounts, O_RDONLY);
}
```

All error messages displayed by **perror** are stored in an array named **sys_errno**, an external array of character strings. The **perror** function uses the variable **errno** as the index to the array element containing the desired message.

Encountering System Errors

This section describes some aspects of system errors and how they relate to user programs. For a list and description of system errors, see **messages** in *XENIX System Reference.*

Programs that encounter serious errors, such as hardware failures or internal errors, generally do not receive detailed reports on the cause of the errors. Instead, PC XENIX treats these errors as system errors. It reports them by displaying a system error message on the system console.

Most system errors occur during calls to system functions. If the system error is recoverable, the system returns an error value to the program and sets the **errno** variable to an appropriate value. No other information about the error is available.

Although the system lets two or more programs share a given resource, it does not keep close track of which program is using the resource at any given time. When an error occurs, the system returns an error value to all programs regardless of which one caused the error. No information about which program caused the error is available.

System errors that occur during routine I/O operations initiated by the system itself generally do not affect user programs. Such errors cause the system to display appropriate system error messages on the terminal.

The system does not detect some system errors until after the corresponding function has returned successfully. Such errors occur when data written to a file by a program has been queued for writing to disk at a more convenient time, or when a portion of data to be read from disk is found to already be in memory and the remaining portion is not read until later. In such cases, the system assumes that the subsequent read or write operations are carried out successfully and passes control back to the program along with a successful return value. If an operation is not carried out successfully, it causes a delayed error.

When a delayed error occurs, the system usually attempts to return an error on the next call to a system function that accesses the same file or resource. If the program has already terminated or does not make a suitable call, the error is not reported.

Using Error Signals

Some program errors cause PC XENIX to generate error signals. These signals are passed back to the program that caused the error and normally terminate the program. The most common error signals are **SIGBUS**, the bus error signal; **SIGFPE**, the floating point exception signal; **SIGSEGV**, the segment violation signal; **SIGSYS**, the system call error signal; and **SIGPIPE**, the pipe error signal. Other signals are described under **signal** in *XENIX System Reference.*

A program can catch an error signal and perform its own error processing by using the **signal** function. This function, as described in Chapter 8, "Using Signals," can set the action of a signal to a user-defined action. For example, the function call:

```
signal(SIGBUS, fixbus);
```

sets the action of the bus error signal to the action defined by the user-supplied function **fixbus**. Such a function usually attempts to remedy the problem, or at least display detailed information about the problem before terminating the program.

For details about how to catch, redefine, and restore these signals, see Chapter 8, "Using Signals."

Appendix A. IBM Personal Computer XENIX C Language

This appendix describes some characteristics of the IBM Personal Computer XENIX C Language compiler. These characteristics can vary on other C language compilers.

Data Types

The following table lists the C language data types and their size characteristics on the IBM Personal Computer XENIX C Language compiler:

Data Type	Bits
char	8
unsigned char	8
short	16
unsigned short	16
int	16
unsigned int	16
long	32
unsigned long	32
float	32
double	64
enum	16
pointer	16 or 32 depending on the program model (small, medium, large, huge) and on if the variable is defined as **near** or **far**

Figure A-1. IBM Personal Computer XENIX C Language Data Type Size Specifications

The **char** and **int** types are signed. The **enum** type is a unique data type.

Structures

Names of structure members do not have to be distinct from structure tags, from names of structure members that are located in different structures, or from names of other variables.

You can assign a structure to another structure that has the same type, and you can pass a structure by value to a function. A function can return a structure.

Bit Fields

Fields are assigned from left to right. You must define bit fields as having type **unsigned int**. You can initialize bit fields.

Unions

You can initialize a **union** by specifying a value for the first member of the union.

Constants

A character constant has type **char** and can contain one character only. In type conversions, character constants undergo sign extension.

The maximum size of an integral or floating point constant is determined by its type. Hexadecimal and octal constants are unsigned.

String Literals

The maximum length of a string, including the ending null character (\0), is 512 bytes.

Escape Sequences

The following table describes the escape sequences recognized by the IBM Personal Computer XENIX C Language compiler:

Escape Sequence	**Character Represented**
\b	Backspace
\f	Form feed (new page)
\n	New-line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Single quotation
\"	Double quotation
\\	Backslash
\0	NUL character (used in the C language to indicate the end of a character string)
\new-line character	Line continuation character (used in C language character strings to indicate the current line continues on the next line)
\octnum	ASCII code for the character indicated by the octal number *octnum*. The octal number can contain one, two, or three digits.
\x*hexnum*	ASCII code for the character indicated by the hexadecimal number *hexnum*. The hexadecimal number can contain one or two digits.

Figure A-2. IBM Personal Computer XENIX C Language Escape Sequences

Reserved Words

The IBM Personal Computer XENIX C Language compiler reserves the following words:

auto
break
case
char
const
continue
default
do
double
else
enum
extern
float
for
goto
if
int
long
register
return
short
sizeof
static
struct
switch
typedef
union
unsigned
void
while

Figure A-3. IBM Personal Computer XENIX C Language Compiler Reserved Words

The following words are reserved by the compiler when you specify certain **cc** command options only:

far
huge
near

Identifiers

The compiler uses the first 31 characters in an identifier (internal or external) as significant characters. Uppercase and lowercase letters are distinguished.

Register Variables

The compiler provides two registers for variables having **char**, **unsigned char**, **int**, **unsigned int**, **short int**, and **unsigned short int** types. Also, variables that are qualified as **near** pointers can share these two registers.

Right Shift Operator

The compiler performs an arithmetic right shift when right shifting a signed variable. The arithmetic right shift fills vacated bits with a copy of the sign bit.

sizeof Operator

In connection with the **sizeof** operator, a byte is an 8-bit quantity.

Pointer Operator

In a pointer expression (involving the pointer operator ->), the left operand must have the same type, or be cast to the same type, as the structure or union to which the right operand belongs.

switch Statements

A **switch** expression must have an integral or **enum** type. The compiler casts each **case** expression to the type of its corresponding **switch** expression.

Data Layout

Within a pair of bytes or each two-byte word, the low-order byte is at the lowest address. Data objects are aligned on word boundaries.

Preprocessor Control Lines

The # character can be preceded by space and tab characters. Space characters and tab characters can also separate the # and the preprocessor keyword. The preprocessor uses the first 31 characters in a preprocessor directive as significant characters.

The IBM Personal Computer XENIX C Language preprocessor recognizes the following keywords:

#defined
#elif
#else
#endif
#if
#if defined
#ifdef
#ifndef
#include
#line
#undef

Figure A-4. IBM Personal Computer XENIX C Language Preprocessor Keywords

The following table summarizes the preprocessor limits:

Preprocessor Construct	Limit
Size of a macro definition	512 bytes
Arguments to a macro definition	32 arguments
Size of a preprocessor argument	256 bytes
Nesting of preprocessor conditional statements	32 levels
Nesting of **include** files	10 levels

Figure A-5. IBM Personal Computer XENIX C Language Preprocessor Limits

Run-Time Limits

The following table summarizes the limits that apply to programs at run time. If a program exceeds one of these limits, an error message may inform you of the problem:

Description	Limit
File size	2^{32} - 1 bytes (4 gigabytes)
Number of open files	20[1] files
Number of characters in the **cc** command (including program name)	128 characters
Size of the environment table	32K bytes

Figure A-6. Run-Time Limits

Other Limitations

The IBM Personal Computer XENIX C Language compiler contains the following additional limitations:

Description	Limit
Size of an array	5 dimensions
Nesting of aggregate initializers	10 levels
Depth of structure nesting	5 levels
Depth of union nesting	5 levels
Level of indirection	5 levels

Figure A-7. IBM Personal Computer XENIX C Language Limits

The compiler does not set explicit limits on the number and complexity of declarations, definitions, and statements in a function or a program. If the compiler encounters a function or a program that is too large or too complex for it to process, the compiler produces an error message to that effect.

---

[1] Five files open automatically (**stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**), leaving 15 files available for a program to open.

Appendix B. A Common Library for PC XENIX and DOS

This appendix lists the library routines that form the Common C Library for the PC XENIX C Compiler and the DOS C Compiler. This appendix also lists variables and include files that are common to PC XENIX and DOS. Use these common routines, variables, and include files to develop C programs for both the PC XENIX and DOS environments. They provide an identical interface to a set of operations that are useful on both PC XENIX and DOS.

Common Routines

The following table contains the common library routines for PC XENIX and DOS:

abort *	dup2	fread *	islower	putw	strncmp
abs	ecvt	free	isprint	qsort	strncpy
access *	execl *	freopen *	ispunct	rand	strpbrk
acos	execle *	frexp	isspace	read *	strrchr
asctime	execlp *	fscanf	isupper	realloc	strspn
asin	execv *	fseek *	isxdigit	rewind	strtok
assert	execve *	fstat *	j0	sbrk	swab
atan	execvp *	ftell *	j1	scanf	system *

- atan2
- atof
- atoi
- atol
- bsearch
- cabs
- calloc
- ceil
- chdir *
- chmod *
- chsize
- clearerr
- close
- cos
- cosh
- creat *
- ctime
- dup
- exit
- exp
- fabs
- fclose
- fcvt
- fdopen
- feof
- ferror
- fflush
- fgetc
- fgets
- fileno
- floor
- fmod
- fopen *
- fprintf
- fputc
- fputs
- ftime *
- fwrite *
- gcvt
- getc
- getchar
- getcwd
- getenv
- getpid *
- gets
- getw
- gmtime
- hypot
- isalnum
- isalpha
- isascii
- iscntrl
- isdigit
- isgraph
- jn
- ldexp
- localtime
- locking *
- log
- log10
- longjmp
- lseek *
- malloc
- mktemp
- modf
- open *
- perror
- pow
- printf
- putc
- putchar
- puts
- setbuf
- setjmp
- signal *
- sin
- sinh
- sprintf
- sqrt
- srand
- sscanf
- stat *
- strcat
- strchr
- strcmp
- strcpy
- strcspn
- strdup
- strlen
- strncat
- tan
- tanh
- time
- toascii
- tolower
- toupper
- tzset
- _tolower
- _toupper
- umask *
- ungetc
- unlink *
- utime *
- write *
- y0
- y1
- yn

Routines marked by an asterisk (*) have a slightly different operation or meaning in the DOS environment than they do under PC XENIX. These differences are described in the following section. Routines that are not marked function the same in DOS as they do in PC XENIX. For complete descriptions, see *XENIX System Reference*.

Differences between Common Routines

The following sections explain how the DOS routines of the common library differ from their PC XENIX counterparts.

abort
The **abort** routine terminates the process and returns control to the operating system without creating a core file.

access
The **access** routine checks the access to a given file. Access does not depend on real and effective IDs as it does in the PC XENIX environment. Under DOS, the real and effective IDs are ignored. The **amode** parameter can be any combination of the values:

```
04  Read
02  Write
00  Check for existence
```

You cannot use the "Execute" access mode (**01**).

The EROFS and ETXTBSY error values are not used.

chdir
The **chdir** routine causes the named directory to become the current working directory in the same way as it does in the PC XENIX environment. The difference is that the directory path name under DOS must use the backslash separator (\) and not the slash (/).

chmod
The **chmod** routine can set the owner access permissions for a given file, but it ignores all other permissions settings. The **mode** parameter can be:

00400	Read by owner
00200	Write by owner
00600	Read and write by owner

If the owner does not have write permission, the file is a read-only file.

Real or effective IDs do not affect the **chmod** routine under DOS. The EPERM and ETXTBSY error values are not used.

creat
The **creat** routine creates a new file or prepares an existing file for writing. When **creat** creates a file, **mode** defines the access permissions. It allows only owner permissions. Real and

effective user and group IDs do not affect ownership of the file. (These are ignored under DOS).

Under DOS the **creat** routine does not use the EROFS and ETXTBSY error values.

As in the PC XENIX environment, we prefer the **open** routine over **creat** when creating or opening files in the DOS environment.

exec
The **execl**, **execle**, **execlp**, **execv**, **execve**, and **execvp** routines do not overlay the calling process as in the PC XENIX environment. Instead, they cause the named process file to be copied into whatever memory is currently available. If there is not enough memory for the new process, the exec routine fails and returns to the calling process. Otherwise, the new process begins execution.

Under DOS, the exec routines do not:

- Use the close-on-exec flag to determine open files for the new process.
- Use the set user and group ID access permissions of the new process file to determine effective user and group IDs.
- Set up signal processing for the new process.
- Disable profiling for the new process (profiling is not allowed under DOS).
- Give the new process attributes inherited from the calling process.
- Use the ETXTBSY error value.

The combined size of all arguments in an exec routine under DOS must not exceed 128 bytes.

fopen, freopen
The **fopen** and **freopen** routines open stream files just as they do in the PC XENIX environment. However, there are the following additional values for the **type** string:

t Opens the file in **text** mode. Opening a file in this mode causes the low-level I/O routines to translate carriage return/line feed (CR-LF) character combinations into a single line feed (LF) on input. Similarly on output, line feeds are translated into CR-LF combinations. For example, the call:

```
fopen("test.dat", "rt");
```

opens a file for reading in **text** mode.

b Opens the file in **binary** mode. This mode suppresses translation.

If **t** or **b** is not in the **type** string, the default mode variable **_fmode** defines the mode. If **_fmode** is 0, the default mode is **text**. If the higher-order bit of **_fmode** is 1, the default mode is **binary**.

fread
The **fread** routine removes carriage returns (CR) in all CR-LF pairs read from the input stream if the stream was opened in **text** mode.

fseek
The **fseek** routine moves the file pointer to the given position just as in the PC XENIX environment. However, because DOS uses the carriage return/line feed (CR-LF) character combinations for new-line characters (PC XENIX uses only an LF), an **fseek** call, which moves the file pointer a specific number of bytes past new-line characters, will not move the pointer to same place in the DOS file as it does in the PC XENIX file. For example, if a file contains the characters:

```
abcdef\n09123
```

where **\n** is the newline character and the file pointer is currently at the letter **a**, then the call:

```
fseek(stream, 8, 1);
```

moves the pointer to the digit 0 if the file is a DOS file, or to the digit 9, if the file is a PC XENIX file.

Some **fseek** calls treat DOS and PC XENIX files the same. For example:

```
fseek(stream, 0, 2);
```

always moves the file pointer to the end of the file.

fstat
The **fstat** routine returns less useful information than the **stat** routine, because DOS does not make as much information available for file descriptors as it does full path names. The **fstat** routine can detect device files, but do not use it with directories. The structure returned by **fstat** has the following members:

st_mode	Always read and write for everyone. The **S_IFCHR** flag is set if this is a device. Otherwise, the **S_IFREG** bit is set.
st_ino	Not used.
st_dev	Either drive number of the disk containing the file, or file descriptor if this file is a device.
st_rdev	Either drive number of the disk containing the file, or file descriptor if this file is a device.
st_nlink	Always 1.
st_uid	Not used.
st_gid	Not used.
st_size	Size of the file in bytes.
st_atime	Time of last modification of file.
st_mtime	Time of last modification of file.
st_ctime	Time of last modification of file.
st_dosattr	Contains DOS file attributes.

ftell
The **ftell** routine gets the current file pointer position, just as in the PC XENIX environment. However, for streams opened in text mode, the value returned by **ftell** may not reflect the physical byte offset, because text mode causes carriage return/line-feed translation. Use the **ftell** routine with the **fseek** routine to remember and return to file locations correctly.

ftime
Unlike the system time on PC XENIX systems, the DOS system time does not include the concept of a default time zone. Instead, **ftime** uses the value of a DOS environment variable named TZ to

determine the time zone. You can set the default time zone by setting the TZ variable. If you do not set TZ, the default time zone corresponds to the Pacific Time Zone.

fwrite
The **fwrite** routine replaces every line-feed (LF) character written to the output stream with a carriage return/line-feed (CR-LF) pair if the stream was opened in **text** mode.

getpid
The **getpid** routine returns a unique number. Although you can use the number to uniquely identify the process, it does not have the same meaning as the process ID returned by **getpid** in the PC XENIX environment.

locking
The DOS and PC XENIX versions of the **locking** routine differ in several respects, as listed below.

- Under DOS, it is not possible to lock a file only against write access; locking a region of a file prevents both reading and writing in that region. This means that setting **LK_RLCK** in the **locking** call is equivalent to setting **LK_LOCK**, and setting **LK_NBRLCK** is equivalent to setting **LK_NBLCK**.
- In DOS, specifying **LK_LOCK** or **LK_RLCK** will not cause a program to wait until the specified region of a file is unlocked. Instead, up to 10 attempts are made to lock the file (one attempt per second). If the lock is still unsuccessful after 10 seconds, the **locking** function returns an error value.

  In PC XENIX, if the first attempt at locking fails, the locking process sleeps (suspends execution) and periodically wakes to attempt the lock again. There is no limit on the number of attempts, and the process can continue indefinitely.
- In DOS, you cannot lock overlapping regions of a file.
- In DOS, if more than one region of a file is locked, you can unlock only one region at a time. That region must correspond to a region that was previously locked. You cannot unlock more than one region at a time, even if the regions are adjacent.

lseek
The **lseek** routine is similar to the **fseek** routine under DOS whenever the given file descriptor has been opened in **text** mode. In other words, **lseek** must move the file pointer one additional byte for each new-line character in the DOS file in order to move the file pointer to the same position in the PC XENIX file.

open
The **open** routine opens a file descriptor for a named file, just as in the PC XENIX environment. However, one additional **oflag** value, O_BINARY, have been added, and two **oflag** values, O_NDELAY and O_SYNCW, have been removed.

The O_BINARY flag causes the file to open in the opposite mode specified by the **_fmode** variable. For example, if the default mode is **text** (**_fmode** is 0), using O_BINARY opens the file in **binary** mode. If you do not use O_BINARY, the file opens in **text** mode.

The EISDIR, EROFS, ETXTBSY, and ENXIO error values are not used.

read
The **read** routine reads characters from the file given by a file descriptor just as in the PC XENIX environment. However, if the file has been opened in **text** mode, **read** replaces each CR-LF pair read from the file with a single LF character. The number of bytes returned is the number of bytes remaining after the CR-LF pairs have been replaced. Thus, the return value may not always correspond with the actual number of bytes read. This is normal and has no implications as far as detecting the end of the file.

signal
The **signal** routine can only handle the SIGINT signal. In DOS, SIGINT is defined to be INT 23H (the Ctrl-C signal).

stat
The **stat** routine returns a structure defining the current status of the given file. The structure members have the following names and meaning:

st_mode	User read/write/execute bits are set. The execute bit is inferred from the file name extension. These are copied into the group and other bits. All files have either the S_IFREG or S_IFDIR bits set.
st_ino	Not used.
st_dev	Drive number of the disk containing the file.
st_rdev	Drive number of the disk containing the file.
st_nlink	Always 1.
st_uid	Not used.
st_gid	Not used.
st_size	Size of the file in bytes.

st_atime	Time of last modification of file.
st_mtime	Time of last modification of file.
st_ctime	Time of last modification of file.
st_dosattr	Contains DOS file attributes.

system
The **system** routine passes the given string to the operating system for execution. To execute this string, the full path name of the directory containing the DOS COMMAND.COM program must be assigned to the COMSPC environment variable, or assigned to the PATH environment variable. The call will return an error if COMMAND.COM cannot be found using these variables.

umask
The **umask** routine can set a mask for owner read and write access permissions only. It ignores all other permissions.

unlink
The **unlink** routine always deletes the given file. Because DOS does not allow multiple links to the same file, unlinking a file is the same as deleting it.

The EBUSY, ETXTBSY, and EROFS error values are not used.

utime
The **utime** routine sets the file modification time only; DOS does not maintain a separate access time.

In case of error, the **EACCES** and **ENOENT** values may be returned for **errno** in DOS. In addition, the **EMFILE** value may be used; under DOS, the file must be opened to set the modification time.

write
The **write** routine writes a specified number of characters to the file named by the given file descriptor just as in the PC XENIX environment. However, if the file has been opened in **text** mode, every LF character in the output is replaced by a CR-LF pair before being written. This does not affect the return value.

DOS-Specific Routines

The routines listed below are available only in the DOS C library. You can use these routines in programs you compile in the PC XENIX environment, but which you execute in the DOS environment. These routines are not available for use in the PC XENIX environment.

The following table contains the DOS-specific routines:

bdos
cgets
cprintf
cputs
cscanf
dosexterr
eof
_exit
fcloseall
fgetchar
filelength
flushall
FP_OFF
FP_SEG
fputchar
getch
getche
inp
int86
int86x
intdos
intdosx
isatty
itoa
kbhit
labs
ltoa
mkdir
movedata
outp
putch
rename
rmdir
segread
setmode
sopen
spawnl
spawnle
spawnlp
spawnv
spawnve
spawnvp
strcmpi
strlwr
strnset
strrev
strset
strupr
tell
ultoa
ungetch

Figure B-1. DOS-specific Routines

Common Variables

The following is a list of system-wide variables used in the run-time library and available in both the DOS and PC XENIX environments. The super user or the PC XENIX kernel sets these variables (with the exception of the **environ** variable). You can reference them, but not change them.

daylight
environ
errno
sys_errlist
sys_nerr
timezone
tzname

The DOS run-time library does not use all values of **errno** available on PC XENIX.

DOS-Specific Variables

The following global variables are available only in the DOS C library:

_doserrno
_fmode
_osmajor
_osminor
_psp

Common Include Files

Structure definitions, return value types, and manifest constants used in the descriptions of some of the common routines may vary from environment to environment and are therefore fully defined in a set of include files for each environment. The following DOS include files are compatible with the PC XENIX include files by the same name:

assert.h
ctype.h
errno.h
fcntl.h
math.h
setjmp.h
signal.h
stdio.h
time.h
sys\elocking.h
sys\estat.h
sys\etimeb.h
sys\etypes.h

DOS-Specific Include Files

The following include files are available only in DOS environments.

conio.h
direct.h
dos.h
io.h
process.h
share.h
stdlib.h
sys\eutime.h
v2tov3.h

Appendix C. Assembler Language Interface

This appendix explains how to use 8086/80286 assembler language routines with C language programs and functions. In particular, it explains how to call assembler language routines from C language programs and how to call C language functions from an assembler language routine.

This assembler language interface is useful for assembler language programmers who wish to see the functions of the standard C library and other C libraries. For more information about PC XENIX assembler language, see the IBM Personal Computer PC XENIX *Macro Assembler Reference.*

C Calling Sequence

To receive values from C language function calls or to pass values to C functions, assembler language routines must follow the C argument passing conventions. C language function calls pass their arguments to the given functions by pushing the value of each argument onto the stack. The call pushes the value of the last argument first and the first argument last. If an argument is an expression, the call computes the expression's value before pushing it onto the stack. The same number of bytes the variable occupied in the main program is pushed onto the stack.

Arguments with **char**, **int**, or **unsigned** type occupy a single word (16 bits) on the stack. Arguments with **long** type occupy a doubleword (32 bits) with the value's high-order word occupying the first word. Arguments with **float** type are converted to double type (64 bits). Note that **char** type arguments are zero-extended to **int** type before being pushed on the stack.

If the name of an array is used as an argument, the value passed to the function is the address of the beginning of the array.

If an argument is a structure, the function call pushes the last word of the structure first and each successive word in turn until the first word is pushed. When you want the called routine to modify an argument (call-by-reference), pass the address of the argument by preceding it with an ampersand (&).

After a function returns control to a routine, the calling routine is responsible for removing arguments from the stack.

Entering an Assembler Routine

Assembler language routines that receive control from C function calls should preserve the contents of the bp, si, and di registers and set the bp register to the current sp register value before proceeding with their tasks. The following example illustrates the recommended instruction sequence for entry to an assembler language routine:

```
entry:
        push    bp
        mov     bp,sp
        push    di
        push    si
```

This is the same sequence used by the C compiler.

If this sequence is used, the last argument passed by the function call (which is also the first argument given in the call's argument list) is at address 4[bp]. Subsequent arguments begin at address 6[bp] or 8[bp], depending on the size of the first argument.

This sequence is strongly recommended even if the si and di registers are not modified, because it allows backtracing with the **adb** program during program debugging.

Return Values

Assembler language routines that wish to return values to a C language program or receive return values from C functions must follow the C return value conventions. C functions place return values that have **int**, **char**, or **unsigned** type in the ax register. They place values with **long** type in the ax and dx registers, with the high-order word in dx.

To return a structure or a floating-point value, C functions place the address of the given value in the ax register. The structure or floating-point value must be in a static area in memory. Long addresses are returned in the ax and dx registers, with the segment selector in dx.

Exiting a Routine

Assembly language routines that return control to C programs should restore the values of the bp, si, and di registers before returning control. The following example illustrates the recommended instruction sequence for exiting a routine:

```
pop     si
pop     di
mov     sp, bp
pop     bp
ret
```

This sequence does not change the ax, bx, cx, or dx registers or any of the segment registers. The sequence does not remove arguments from the stack. This is the responsibility of the calling routine.

Program Example

To illustrate the assembly language interface, consider the following example of a C function:

```
add(i,j)
int i, j;
{
   return(i+j);
}
```

If written as an assembly language routine, this function must:

- Save the proper registers
- Retrieve the arguments from the stack
- Add the arguments
- Place the return value in the ax register
- Restore registers
- Return control.

The following fragment is an example of the routine can be written:

```
        PUBLIC   _add
_add:   PROC     NEAR
        push     bp
        mov      bp,sp
        push     di
        push     si

        mov      ax,[bp+4]
        add      ax,[bp+6]

        pop      si
        pop      di
        mov      sp, bp
        pop      bp
        ret
_add    endp
```

If an assembly language routine calls this C function:

```
add(i,j)
int i, j;
{
   return (i+j);
}
```

the routine must contain instructions that push the arguments on the stack in the proper order, call the function, and clear the stack. It may then use the return value in the ax register. The following fragment is an example of the instructions that can do this:

```
        PUBLIC   _main
_main   PROC NEAR
        push     bp
        mov      bp,sp
        push     <j value>
        push     <i value>
        call     _add
        mov      sp,bp
        pop      bp
        ret
_main   ENDP
```

Note: The order in which you push and pop parameters on the stack is critical.

Appendix D. C Compiler Messages

This appendix lists the messages displayed by the **cc** command when it encounters errors during compilation of a program. The error messages produced by the C compiler fall into five categories:

- Compiler internal error messages
- Command line error messages
- Warning messages
- Fatal error messages
- Compilation error messages.

Compiler internal error messages indicate an error on the part of the compiler rather than your program. No matter what your source program contains, these messages should not appear. Although these errors are not the fault of your program, you will probably want to rearrange your code so that the program can be compiled.

Command line error messages give you information about invalid or inconsistent command line options. If possible, the compiler continues operation, printing a warning message to indicate which command line options are in effect and which are disregarded. In some cases, command line errors are fatal, and the compiler terminates processing.

Warning messages are informational only; they do not prevent compilation and linking. You can control the level of warnings generated by the compiler by using the **-W** or **-w** options, described in Chapter 1, "cc: A C Compiler." The following list of warning messages includes a number for each message indicating the minimum level that must be set for the message to appear.

Fatal error messages indicate a severe problem, one that prevents the compiler from processing your program. After printing out a message about the fatal error, the compiler terminates without producing an object file or checking for further errors.

Compilation error messages identify actual program errors. No object file is produced for a source file that has such errors. When the compiler encounters a nonfatal program error, it attempts to recover from the error. If possible, the compiler continues to process the source file and produce error messages. If errors are too numerous or too severe, the compiler terminates processing.

Error messages in the warning, fatal, and compilation error message categories have the same basic form:

filename (*linenumber*) : *msg-type errornumber*: *message*

- The *filename* is the name of the source file being compiled.
- The *linenumber* identifies the line of the file containing the error.
- The *msg-type* is **warning**, **fatal**, **error**, or **command line**.
- The *errornumber* is the number associated with the error
- The *message* is a description of the error or warning.

The messages for each category are listed below in numerical order, along with a brief explanation of each error. To look up an error message, first determine the message category, then find the error number.

The section, "Compiler Limits," summarizes limits imposed by the C Compiler (for example, the maximum size of a macro definition).

Note: Sometimes, the text of an error message will make reference to either a DOS switch or file name. These error messages are provided for cross-development purposes. They only occur in the DOS environment.

Compiler Internal Error Messages

The following messages are generated in the event of an internal compiler error:

error 0: UNKNOWN ERROR

Explanation: The compiler has detected an unforeseen error condition.

error 124: CODE GENERATION ERROR

Explanation: The compiler could not generate code for an expression. Usually this occurs with a complex expression; try rearranging the expression.

fatal error 1: assertion count exceeds 5; stopping compilation

Explanation: More than five assertion errors have accumulated, and the compiler cannot continue processing.

Command Line Messages

The following messages indicate errors on the command line used to invoke the compiler. If possible, the compiler continues operation, printing a warning message. In some cases, command line errors are fatal, and the compiler terminates processing.

Command line error 0

Explanation: The compiler has detected an unforeseen error condition. Please report this condition to IBM.

Command line error 1: too many symbols predefined with -D

Explanation: The limit on command line definitions is 11.

Command line error 2: listing has precedence over assembly output

Explanation: Two different listing options were chosen; the assembly listing is not created.

Command line error 3: a previously defined model specification has been overridden

Explanation: Two different memory models are specified; the model specified later is used.

Command line error 5: only one memory model allowed

Explanation: You must choose one memory model; you cannot specify more than one.

Command line error 6: missing source file name

Explanation: You must give the name of the source file you want to compile.

Command line error 7: too many commas

Explanation: Too many commas appear on the command line.

Command line error 8: comma needed before *filename*

Explanation: The fields in the command line must be set off by commas.

Command line error 9: a file name (not a path name) is required

Explanation: The name of a directory is given where the name of a file is required.

Command line warning 10: ignoring unknown flag *string*

Explanation: One of the options given on the command line is not recognized and is ignored.

Command line error 12: too many *option* flags, *string*

Explanation: Too many letters are given with a specific option.

Command line error 13: unknown option (*c*) in *option*

Explanation: One of the letters in the given option is not recognized.

Command line error 14: argument list for *name* too big

Explanation: The combined length of all arguments on the command line (including the program name) may not exceed 128 bytes.

Command line error 15: 80186/286 selected over 8086 for code generation

Explanation: You have specified both the **-M0** and **-M2** options; **-M2** takes precedence.

Command line error 17: unknown floating point option

Explanation: The specified floating-point option is not one of the five valid options.

Command line error 18: only one floating point model allowed

Explanation: You can only give one of the five floating-point options on the command line.

Command line error 19: could not execute *pass*

Explanation: The specified compiler file could not be found.

Command line error 26: unknown -M substring '*string*'

Explanation: A string given with the **-M** option is not recognized.

Command line error 28: -ND not allowed with -Ad

Explanation: The two options conflict: **-Ad** inhibits loading the named segment into DS upon entry to the module.

Command line error 29: -ND not allowed with -Aw

Explanation: The two options conflict: **-Aw** inhibits loading the named segment into DS upon entry to the module.

Command line warning 31: threshold only for far/huge data, ignored

Explanation: The data threshold size given with option **-Mt** is applicable only in large or huge model programs, for allocating data items to the default segment.

Command line fatal error 22623: no more processes, try later

Explanation: The system process table is full. It will not let a new compile process start.

Warning Error Messages

The messages listed in this section indicate potential problems but do not hinder compilation and linking. The number in square brackets ([ ]) at the end of each message gives the minimum warning level that must be set for the message to appear.

warning 1: macro *identifier* requires parameters [1]

Explanation: The given *identifier* was defined as a macro taking one or more arguments, but the identifier is used in the program without arguments.

warning 2: too many actual parameters for macro *identifier* [1]

Explanation: The number of arguments specified with an identifier is greater than the number of formal parameters given in the macro definition of the identifier.

warning 3: not enough actual parameters for macro *identifier* [1]

Explanation: The number of arguments specified with an identifier is less than the number of formal parameters given in the macro definition of the identifier.

warning 4: missing close parenthesis after 'defined' [1]

Explanation: The closing parenthesis is missing from an **#if defined** phrase.

warning 5: *identifier* : redefinition [1]

Explanation: The given *identifier* is redefined.

warning 6: #undef expected an identifier [1]

Explanation: You must give the name of the identifier whose definition you want to remove with the **#undef** directive.

warning 7: unmatched close comment '*/' [1]

Explanation: A comment is started (with '/*') but is not closed (with '*/').

warning 8: newline in string constant [1]

Explanation: A new-line character is not preceded by an escape character (\\) in a string constant.

warning 9: string too big, trailing chars truncated [1]

Explanation: A string exceeds the compiler limit on string size. To correct this problem, you must break the string into two or more strings.

warning 10: illegal null char [1]

Explanation: The single quotes delimiting a character constant must contain one character. For example, the declaration `char a = ' '` is illegal. To represent a null character constant, use an escape sequence (for example, '\0').

warning 11: identifier truncated to "*identifier*" [1]

Explanation: Only the first 31 characters of an identifier are significant.

warning 13: constant too big [1]

Explanation: You lost information because a constant value is too large to be represented in the type to which you assigned it.

warning 14: ***identifier*** **: bitfield type must be unsigned [1]**

Explanation: Bit fields must be declared as **unsigned** integral types. A conversion has been supplied.

warning 15: ***identifier*** **: bitfield type must be integral [1]**

Explanation: Bit fields must be declared as **unsigned** integral types. A conversion has been supplied.

warning 16: ***identifier*** **: no function return type [2]**

Explanation: The return type is missing from a function declaration; the default return type will be **int**.

warning 17: cast of int expression to far pointer [1]

Explanation: A **far** pointer represents a full segmented address. On an 8086/8088 processor, casting an **int** value to a **far** pointer produces an address with a meaningless segment value.

warning 18: ***identifier*** **: uses undefined struct/union** ***identifier*** **[2]**

Explanation: The name of a structure or union type is used before the type is defined.

warning 19: '*identifier*' : unknown size [1]

Explanation: The size of the named variable is not specified.

warning 20: too many actual parameters [1]

Explanation: The number of arguments specified in a function call is greater than the number of parameters specified in the argument type list or in the function definition.

warning 21: too few actual parameters [1]

Explanation: The number of arguments specified in a function call is fewer than the number of parameters specified in the argument type list or in the function definition.

warning 22: pointer mismatch: parameter *n* [1]

Explanation: The given parameter has a different pointer type than is specified in the argument type list or the function definition.

warning 23: parameter *n*'s type is not in union *identifier* [2]

Explanation: The declaration of the given parameter specifies a **union** type, but the parameter's type does not correspond to the type of any of the union members.

warning 24: different types : parameter *n* [2]

Explanation: The type of the given parameter in a function call does not agree with the argument type list or the function definition.

warning 25: function declaration specified variable args [1]

Explanation: The argument type list in a function declaration ends with a comma, indicating that the function can take a variable number of arguments, but no formal parameters for the function are declared.

warning 26: function was declared with formal arguments [1]

Explanation: The function was declared to take arguments, but the function definition does not declare formal parameters.

warning 27: function was declared without formal argument list [1]

Explanation: The function was declared to take no argument (the argument type list consists of the word **void**), but formal parameters are declared in the function definition or arguments are given in a call to the function.

warning 28: parameter *n* declaration different [2]

Explanation: The type of the given parameter does not agree with the corresponding type in the argument type list or with the corresponding formal parameter.

warning 29: declared parameter list differs from definition [1]

Explanation: The argument type list given in a function declaration does not agree with the types of the formal parameters given in the function definition.

warning 30: first parameter list is longer than the second [1]

Explanation: A function is declared more than once and the argument type lists in the declarations differ.

warning 31: second parameter list is longer than the first [1]

Explanation: A function is declared more than once, and the argument type lists in the declarations differ.

warning 32: unnamed struct/union as parameter [1]

Explanation: The structure or union type being passed as an argument is not named, so the declaration of the formal parameter cannot use the name and must declare the type.

warning 33: function must return a value [2]

Explanation: A function is expected to return a value unless it is declared as **void**.

warning 34: sizeof returns 0 [2]

Explanation: The **sizeof** operator is applied to an operand that yields a size of zero.

warning 35: no return value [2]

Explanation: A function declared to return a value does not do so.

warning 36: unexpected formal parameter list [1]

Explanation: A formal parameter list is given in a function declaration and is ignored.

warning 37: '*identifier*' : formal parameters ignored [1]

Explanation: Formal parameters appeared in a function declaration, for example:

```
extern int *f(a,b,c);
```

The formal parameters are ignored.

warning 38: *identifier* : formal parameter has bad storage class [1]

Explanation: Formal parameters must have **auto** or **register** storage class.

warning 39: '*identifier*' : function used as an argument [1]

Explanation: A formal parameter to a function was declared to be a function, which is illegal. The formal parameter is converted to a function pointer.

warning 40: near/far/huge on *identifier* ignored [1]

Explanation: The **near**, **far**, and **huge** keywords have no effect in the declaration of the given *identifier* and are ignored.

warning 41: formal parameter *identifier* is redefined [1]

Explanation: The given formal parameter is redefined in the function body, making the corresponding argument unavailable in the function.

warning 42: '*identifier*' : has bad storage class [1]

Explanation: The specified storage class cannot be used in this context (for example, function parameters cannot be given **extern** class). The default storage class for that context is used in place of the illegal class.

warning 43: '*identifier*' : void type changed to int [1]

Explanation: Only functions may be declared to have **void** type.

warning 44: huge on '*identifier*' ignored, must be an array [1]

Explanation: The **huge** keyword can only be used in array declarations.

warning 45: '*identifier*' : array bounds overflow [1]

Explanation: Too many initializers are present for the given array. The excess initializers are ignored.

warning 46: '&' on function/array, ignored [1]

Explanation: You cannot apply the address-of operator to a function or array identifier.

warning 47: '*operator*' : different levels of indirection [1]

Explanation: An expression involving the specified operator has inconsistent levels of indirection. For example,

```
char **p;
char *q;
.
.
.
p=q;  /* different levels of indirection */
```

warning 48: array's declared subscripts different [1]

Explanation: An array is declared twice with differing sizes. The larger size is used.

warning 49: ***operator*** **: indirection to different types [1]**

Explanation: The indirection operator (*) is used in an expression to access values of different types.

warning 50: strong type mis-match [2]

Explanation: Two different but compatible types are used: for example, a **typedef** type with a non-**typedef** type, or two different but equivalent **struct** or **union** types.

warning 51: data conversion [3]

Explanation: Two data items in an expression had different types, causing the type of one item to be converted.

warning 52: different enum types [1]

Explanation: Two different **enum** types are used in an expression.

warning 53: at least one void operand [1]

Explanation: An expression with type **void** is used as an operand.

warning 54: '*operator*' : illegal with enums [1]

Explanation: You may not use the given operator with an **enum** value. The **enum** value is converted to **int** type.

warning 55: type following '*keyword*' is illegal, ignored [1]

Explanation: An illegal combination occurs (for example, **unsigned float**).

warning 56: overflow in constant arithmetic [1]

Explanation: The result of an operation exceeds 0x7FFFFFFF.

warning 57: overflow in constant multiplication [1]

Explanation: The result of an operation exceeds 0x7FFFFFFF.

warning 59: conversion lost segment [1]

Explanation: The conversion of a **far** pointer (a full segmented address) to a **near** pointer (a segment offset) resulted in the loss of the segment address.

warning 60: conversion of a long address to a short address [1]

Explanation: The conversion of a long address (a 32-bit pointer) to a short address (a 16-bit pointer) resulted in the loss of the segment address.

warning 61: long/short mismatch in arguments : conversion supplied [1]

Explanation: An integral type was assigned to an integer of a different size, causing a conversion to take place. For example, a **long** was given where a **short** was declared.

warning 62: near/far mismatch in arguments: conversion supplied [1]

Explanation: A pointer was assigned to a pointer with a different size, resulting in the loss of a segment address from a **far** pointer or the addition of a segment address to a **near** pointer.

warning 63: 'function' ***identifier*** **too large for post-optimizer [0]**

Explanation: The named function was not optimized because insufficient space was available. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning 64: procedure too large, skipping ***loop inversion or branch sequence or cross jump*** **optimization and continuing [0]**

Explanation: Some optimizations for a function were skipped because insufficient space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning 65: recoverable heap overflow in post optimizer - some optimizations may be missed [0]

Explanation: Some optimizations were skipped because insufficient space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

Fatal Error Messages

The following messages identify fatal errors. The compiler cannot recover from a fatal error; it terminates after printing the error message.

fatal error 1: assertion count exceeds 5; stopping compilation

Explanation: More than five assertion errors have accumulated, and the compiler cannot continue processing.

fatal error 2: out of heap space

Explanation: The compiler has run out of dynamic memory space. This usually means that your program has many symbols and complex expressions. To correct the problem, break down the file into several smaller source files.

fatal error 3: error count exceeds *n*; stopping compilation

Explanation: Errors in the program are too numerous or too severe to allow recovery, and the compiler must terminate.

fatal error 4: unexpected EOF

Explanation: You have insufficient space on the default disk drive for the compiler to create the temporary files it needs. The space required is approximately two times the size of the source file.

fatal error 5: string too big for buffer

Explanation: An input source line exceeded the buffer space of the compiler.

fatal error 6: write error on compiler intermediate file

Explanation: The compiler was unable to create the intermediate files used in the compilation process. The exact reason is unknown.

fatal error 7: unrecognized flag *string* in *pass*

Explanation: One of the options given on the command line was not recognized, and the file cannot be processed.

fatal error 8: no input file specified

Explanation: You must give at least one source file as input to the compiler.

fatal error 9: compiler limit : possibly a recursively defined macro

Explanation: The expansion of a macro exceeded the available space. Check to see whether the macro is recursively defined or if the expanded text is too large.

fatal error 10: compiler limit : macro expansion too big

Explanation: The expansion of a macro exceeds the available space.

fatal error 11: recursively defined macro *identifier*

Explanation: The given *identifier* was defined recursively.

fatal error 12: bad parenthesis nesting

Explanation: The parentheses in a preprocessor directive are not matched.

fatal error 13: cannot open *filename*

Explanation: The given file cannot be opened, possibly because you do not have read permission.

fatal error 14: too many include files

Explanation: Nesting of **#include** directives exceeds the limit of 10 levels.

fatal error 15: cannot find *filename*

Explanation: The given file does not exist or cannot be found. Check to make sure your environment settings are valid and that you have given the correct path name for the file.

fatal error 16: #if[n]def expected an identifier

Explanation: You must specify an identifier with the **#ifdef** and **#ifndef** directives.

fatal error 17: invalid integer constant expression

Explanation: The expression in an **#if** directive must evaluate to a constant.

fatal error 18: unexpected '#elif'

Explanation: The **#elif** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** directive.

fatal error 19: unexpected '#else'

Explanation: The **#else** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** directive.

fatal error 20: unexpected '#endif'

Explanation: An **#endif** directive appears without a matching **#if**, **#ifdef**, or **#ifndef** directive.

fatal error 21: bad preprocessor command '*string*'

Explanation: The characters following the number sign (#) do not form a preprocessor directive.

fatal error 22: expected '#endif'

Explanation: An **#if**, **#ifdef**, or **#ifndef** directive was not terminated with an **#endif** directive.

fatal error 26: parser stack overflow, please simplify your program

Explanation: Your program cannot be processed because the space required to parse the program causes a stack overflow in the compiler. To solve this problem, try to simplify your program.

fatal error 27: DGROUP data allocation exceeds 64K

Explanation: Allocation of variables to the default data segment exceeds 64K.

fatal error 32: cannot open listing file *filename*

Explanation: The file name or path name given for the listing file is not valid.

fatal error 33: cannot open assembly language output file *filename*

Explanation: The file name or path name given for the assembly language output file is not valid.

fatal error 34: cannot open source file *filename*

Explanation: The file name or path name given for the source file is not valid.

fatal error 35: expression too complex, please simplify

Explanation: The compiler is unable to parse an input expression.

fatal error 41: cannot open compiler intermediate file

Explanation: The compiler is unable to create intermediate files used in the compilation process because no more file handles are available.

fatal error 42: cannot open compiler intermediate file - no such file or directory

Explanation: The compiler is unable to create intermediate files used in the compilation process.

fatal error 43: cannot open compiler intermediate file

Explanation: The compiler is unable to create intermediate files used in the compilation process. The exact reason is unknown.

fatal error 44: out of disk space for compiler intermediate file

Explanation: The compiler is unable to create intermediate files used in the compilation process because no more space is available. To correct the problem, make more space available on the disk and recompile.

Compilation Error Messages

The messages listed below indicate that your program has errors. When the compiler encounters any of the errors listed in this section, it continues parsing the program (if possible) and outputs additional error messages. However, no object file is produced.

error 0: UNKNOWN ERROR

Explanation: An unforeseen error condition has been detected by the compiler.

error 1: newline in constant

Explanation: A new-line character in a character or string constant must be preceded by the backslash escape character (\).

error 2: out of macro actual parameter space

Explanation: Arguments to preprocessor macros may not exceed 256 bytes.

error 3: missing open paren after keyword ‘defined’

Explanation: Parentheses must surround the identifier you want to check in an **#if** directive.

error 4: expected ‘defined(id)’

Explanation: An **#if** directive has a syntax error.

error 5: #line expected a line number

Explanation: A **#line** directive lacks the mandatory line number specification.

error 6: #include expected a file name

Explanation: An **#include** directive lacks the mandatory file name specification.

error 7: #define syntax

Explanation: A **#define** directive has a syntax error.

error 8: ‘*c*’ :unexpected in macro definition

Explanation: The character *c* is misused in a macro definition.

error 9: reuse of macro formal *identifier*

Explanation: The parameter list in a macro definition contains two occurrences of the same identifier.

error 10: ‘*c*’ : unexpected in formal list

Explanation: The character *c* is misused in a macro definition′s list of formal parameters.

error 11: ‘*identifier*’ : definition too big

Explanation: Macro definitions may not exceed 256 bytes.

error 12: missing name following ‘<’

Explanation: An **#include** directive lacks the mandatory filename specification.

error 13: missing ’>’

Explanation: The closing angle bracket (‘>’) is missing from an **#include** directive.

error 14: preprocessor command must start as first non-whitespace

Explanation: Non-whitespace characters appear before the number sign (#) of a preprocessor directive on the same line.

error 15: too many chars in constant

Explanation: A character constant is limited to a single character or escape sequence. (Multicharacter character constants are not supported.)

error 16: no closing single quote

Explanation: A character constant did not end with ‘’’.

error 17: illegal escape sequence

Explanation: The characters after the escape character (\) do not form a valid escape sequence.

error 18: unknown character '0x*n*'

Explanation: The given hexadecimal number does not correspond to a character.

error 19: expected preprocessor command, found '*c*'

Explanation: The character following a number sign (#) is not the first letter of a preprocessor directive.

error 20: bad octal number '*n*'

Explanation: The character *n* is not a valid octal digit.

error 21: expected exponent value, not '*n*'

Explanation: The exponent of a floating-point constant is not a valid number.

error 22: '*n*' : too big for char

Explanation: The number *n* is too large to be represented as a character.

error 23: divide by 0

Explanation: The second operand in a division operation (/) evaluates to zero, giving undefined results.

error 24: mod by 0

Explanation: The second operand in a remainder operation (%) evaluates to zero, giving undefined results.

error 25: '*identifier*' : enum/struct/union type redefinition

Explanation: The given *identifier* has already been used for an enumeration, structure, or union tag.

error 26: '*identifier*' : member of enum redefinition

Explanation: The given *identifier* has already been used for an enumeration constant, either within the same enumeration type or within another enumeration type with the same visibility.

error 27: compiler limit : struct/union nesting

Explanation: Nesting of structure and union definitions may not exceed five levels.

error 28: struct/union member needs to be inside a struct/union

Explanation: Structure and union members must be declared within the structure or union.

error 29: '*identifier*' : fields only in structs

Explanation: Only structure types may contain bit fields.

error 30: struct/union member redefinition

Explanation: The same identifier was used for more than one structure or union member.

error 31: '*identifier*' : function cannot be struct/union member

Explanation: A function cannot be a member of a structure; use a pointer to a function instead.

error 32: '*identifier*' : base type with near/far/huge not allowed

Explanation: Declarations of structure and union members may not use the **near**, **far**, and **huge** keywords.

error 33: '*identifier*' : field has indirection

Explanation: The bit field is declared as a pointer (*), which is not allowed.

error 34: '*identifier*' : field type too small for number of bits

Explanation: The number of bits specified in the bit field declaration exceeds the number of bits in the given **unsigned** type.

error 35: struct/union '*identifier*' : unknown size

Explanation: A member of a structure or union has an undefined size.

error 36: left of '->*identifier*' or '.*identifier*' must have struct/union type

Explanation: The expression before the member selection operator '->' is not a pointer to a structure or union type, or the expression before the member selection operator '.' does not evaluate to a structure or union.

error 37: left of '->' or '.' specifies undefined struct/union '*identifier*'

Explanation: The expression before the member selection operator '->' or '.' identifies a structure or union type that is not defined.

error 38: '*identifier*' : not struct/union member

Explanation: The given *identifier* is used in a context that requires a structure or union member.

error 39: '->' requires struct/union pointer

Explanation: The expression before the member selection operator '->' is not a pointer to a structure or union.

error 40: '.' requires struct/union name

Explanation: The expression before the member selection operator '.' is not the name of a structure or union.

error 41: keyword 'enum' illegal

Explanation: The **enum** keyword appears in a structure or union declaration, or an **enum** type definition is not formed correctly.

error 42: keyword 'enum' required

Explanation: The **enum** keyword is required in declarations of enumeration types.

error 43: illegal break

Explanation: A **break** statement is legal only when it appears within a **do**, **for**, **while**, or **switch** statement.

error 44: illegal continue

Explanation: A **continue** statement is legal only when it appears within a **do**, **for**, or **while** statement.

error 45: '*identifier*' : label redefined

Explanation: The given *identifier* appears before more than one statement in the same function.

error 46: illegal case

Explanation: The **case** keyword may only appear within a **switch** statement.

error 47: illegal default

Explanation: The **default** keyword may only appear within a **switch** statement.

error 48: more than one default

Explanation: A **switch** statement contains too many **default** labels (only one is allowed).

error 49: cast has illegal formal parameter list

Explanation: A formal parameter list is given in a type cast expression.

error 50: non-integral switch expression

Explanation: Switch expressions must be integral.

error 51: case expression not constant

Explanation: Case expressions must be integral constants.

error 52: case expression not integral

Explanation: Case expressions must be integral constants.

error 53: case value '*n*' already used

Explanation: The case value *n* has already been used in this **switch** statement.

error 54: expected '(' to follow '*identifier*'

Explanation: The context requires parentheses after the function *identifier*.

error 55: expected formal parameter list, not a type list

Explanation: An argument type list appears in a function definition instead of a formal parameter list.

error 56: illegal expression

Explanation: An expression is illegal because of a previous error. (The previous error may not have produced an error message.)

error 57: expected constant expression

Explanation: The context requires a constant expression.

error 58: constant expression is not integral

Explanation: The context requires an integral constant expression.

error 59: syntax error : '*token*'

Explanation: The given *token* caused a syntax error.

error 60: syntax error : EOF

Explanation: The end of the file was encountered unexpectedly, causing a syntax error.

error 61: syntax error : identifier '*identifier*'

Explanation: The given *identifier* caused a syntax error.

error 62: type '*identifier*' unexpected

Explanation: The given type is misused.

error 63: '*identifier*' : not a function

Explanation: The given *identifier* was not declared as a function, but an attempt was made to use it as a function.

error 64: term does not evaluate to a function

Explanation: An attempt was made to call a function through an expression that does not evaluate to a function pointer.

error 65: '*identifier*' : undefined

Explanation: The given *identifier* is not defined.

error 66: cast to function returning . . . is illegal

Explanation: An object cannot be cast to a function type.

error 67: cast to array type is illegal

Explanation: An object cannot be cast to an array type.

error 68: illegal cast

Explanation: A type used in a cast operation is not a legal type.

error 69: cast of 'void' term to non-void

Explanation: The **void** type may not be cast to any other type.

error 70: illegal sizeof operand

Explanation: The operand of a **sizeof** expression must be an identifier or a type name.

error 71: '*class*' : bad storage class

Explanation: The given storage *class* cannot be used in this context.

error 72: '*identifier*': initialization of a function

Explanation: Functions may not be initialized.

error 73: '*identifier*' : cannot initialize array in function

Explanation: Arrays can only be initialized at the external level.

error 74: cannot initialize struct/union in function

Explanation: Structures and unions can only be initialized at the external level.

error 75: '*identifier*' : array initialization needs curly braces

Explanation: The braces ({ }) around an array initializer are missing.

error 76: struct/union initialization needs curly braces

Explanation: The braces ({ }) around a structure or union initializer are missing.

error 77: non-integral field initializer *identifier*

Explanation: An attempt is made to initialize a bitfield member of a structure with a non-integral value.

error 78: too many initializers

Explanation: The number of initializers exceeds the number of objects to be initialized.

error 79: '*identifier*' is an undefined struct/union

Explanation: The given *identifier* is declared as a structure or union type that has not been defined.

error 80: '*expression*' was the use of the struct/union

Explanation: An undefined structure or union type variable is used in the given *expression*.

error 81: compiler limit : initializers too deeply nested

Explanation: The compiler limit on nesting of initializers has been exceeded. The limit ranges from 10 to 15 levels, depending on the combination of types being initialized. To correct this problem, simplify the data type being initialized to reduce the levels of nesting, or assign initial values in separate statements after the declaration.

error 82: redefinition of formal parameter '*identifier*'

Explanation: A formal parameter to a function is redeclared within the function body.

error 83: array '*identifier*' already has a size

Explanation: The dimensions of the given array have already been declared.

error 84: function '*identifier*' already has a body

Explanation: The given function has already been defined.

error 85: '*string*' : ignored

Explanation: The given text appeared out of context and was ignored.

error 86: '*identifier*' : redefinition

Explanation: The given *identifier* was defined more than once.

error 87: '*identifier*' : missing subscript

Explanation: To reference an element of an array, you must use a subscript.

error 88: use of undefined struct/union '*identifier*'

Explanation: The given *identifier* was used to refer to a structure or union type that is not defined.

error 89: typedef specifies a near/far function

Explanation: The **near** or **far** keyword is used in a **typedef** declaration.

error 90: function returns array

Explanation: A function may not return an array. (It may return a pointer to an array.)

error 91: function returns function

Explanation: A function may not return a function. (It may return a pointer to a function.)

error 92: array element type cannot be function

Explanation: Arrays of functions are not allowed.

error 94: label '*identifier*' was undefined.

Explanation: The function does not contain a statement labeled with the given *identifier*.

error 95: parameter has type void

Explanation: Formal parameters and arguments to functions may not have **void** type.

error 96: struct/union comparison illegal

Explanation: You cannot compare two structures or unions. (You can, however, compare individual members of structures and unions.)

error 97: illegal initialization

Explanation: An initialization is illegal because of a previous error. (The previous error may not have produced an error message.)

error 98: non-address expression

Explanation: An attempt was made to initialize an item that is not an lvalue.

error 99: non-constant offset

Explanation: An initializer uses a non-constant offset.

error 100: illegal indirection.

Explanation: The indirection operator (*) was applied to a nonpointer value.

error 101: '&' on constant

Explanation: Only variables and functions can have their address taken.

error 102: '&' requires lvalue

Explanation: The address-of operator can only be applied to lvalue expressions.

error 103: '&' on register variable

Explanation: Register variables cannot have their address taken.

error 104: '&' on bit field ignored

Explanation: Bit fields cannot have their address taken.

error 105: ‘*operator*’ needs lvalue.

Explanation: The given *operator* must have an lvalue operand.

error 106: ‘*operator*’ : left operand must be lvalue

Explanation: The left operand of the given *operator* must be an lvalue.

error 107: illegal index, indirection not allowed

Explanation: A subscript was applied to an expression that does not evaluate to a pointer.

error 108: non-integral index

Explanation: Only integral expressions are allowed in array subscripts.

error 109: subscript on non-array.

Explanation: A subscript was used on a variable that is not an array.

error 110: ‘+’ : 2 pointers

Explanation: Two pointers may not be added.

error 111: pointer + non-integral value

Explanation: Only integral values may be added to pointers.

error 112: illegal pointer subtraction

Explanation: Only pointers that point to the same type may be subtracted.

error 113: '-' : right operand pointer

Explanation: The right operand in a subtraction operation (-) is a pointer, but the left operand is not.

error 114: '*operator*' : pointer on left; needs integral right

Explanation: The left operand of the given *operator* is a pointer; the right operand must be an integral value.

error 115: '*operator*' : incompatible types

Explanation: An expression contains types that are not compatible.

error 116: '*operator*' : bad left *or* right operand

Explanation: The specified operand of the given *operator* is an illegal value.

error 117: '*operator*' : illegal for struct/union

Explanation: Structure and union type values are not allowed with the given *operator*.

error 118: negative subscript

Explanation: A value defining an array size was negative.

error 119: ‘typedefs’ both define indirection

Explanation: Two **typedef** types were used to declare an item and both **typedef** types have indirection. For example, the declaration of *p* in the following example is illegal.

```
typedef int *P_INT;
typedef short *P_SHORT;
/* this declaration is illegal */
P_SHORT P_INT p;
```

error 120: ‘void’ illegal with all types

Explanation: The **void** type cannot be used in operations with other types.

error 121: typedef specifies different enum

Explanation: Two different enumeration types defined with **typedef** were used to declare an item, but the enumeration types are different.

error 122: typedef specifies different struct

Explanation: Two structure types defined with **typedef** were used to declare an item, but the structure types are different.

error 123: typedef specifies different union

Explanation: Two union types defined with **typedef** were used to declare an item, but the union types are different.

error 124: CODE GENERATION ERROR

Explanation: The compiler could not generate code for an expression. Usually this occurs with a complex expression. Try rearranging the expression.

error 125: allocation exceeds 64K for *identifier*

Explanation: The given item exceeds the limit of 64K bytes. The only items that are allowed to exceed 64K bytes are **huge** arrays.

error 126: auto allocation exceeds 32K

Explanation: The space allocated for the local variables of a function exceeds the limit of 32K bytes.

error 127: parameter allocation exceeds 32K

Explanation: The storage space required for the parameters to a function exceeds the limit of 32K bytes.

error 128: huge '*identifier*' cannot be aligned to segment boundary

Explanation: The given array violates one of the restrictions imposed on **huge** arrays; review the discussion of these restrictions for details.

error 129: static procedure '*identifier*' not found

Explanation: A forward reference was made to a missing static procedure.

error 130: #line expected a string containing the file name

Explanation: A **#line** directive specified an invalid file name.

error 131: attributes specify more than one near/far/huge

Explanation: Conflicting near, far, or huge keywords were implied in a pointer declaration.

error 132: syntax error: unexpected identifier

Explanation: An identifier appeared in an invalid position in the statement.

error 133: array '*identifier*': unknown size

Explanation: An element of an array had an undefined size.

Run-Time Library Error Messages

The following message may be generated at run time when your program has a serious error.

Note: Attempting to divide an integer by 0 is an error. However, unless you make provisions to trap the signal, this error will be ignored. The error signal is 04, SIGILL, illegal instruction.

error 2000: Stack overflow

Explanation: Your program has run out of stack space. This can occur when a program uses a large amount of local data or is heavily recursive. The program is terminated with an exit status of 255. To correct the problem, allocate a larger stack size by using the **-F** option on the **cc** command line.

Index

Special Characters

.c file 1-2
.i file 1-18
.L file 1-14
.o file 1-4, 1-9
.s file 1-14, 1-20
-c option
-C option
-i option
-I option
-l option
-L option
-o option
-O option
-p option
-P option
-s option
-S option
-w option
-W option
-X option
/etc/termcap file 4-3
/usr/include files 2-1
/usr/include/sys files 2-6

A

a.out, default output file 1-2
abort routine B-3
access routine B-3
addch function 4-13
adding characters 4-13, 4-24
adding strings 4-14, 4-24
addstr function 4-14
advanced options, using 1-20
argc, argument count variable 3-2
argv, argument value variable 3-2
assembler language interface C-1
assembler language listing, -L option 1-14
assembler routine entry C-2
assembler routine exit C-3
assembler source files 1-20
assembler source listing, -S option 1-14

B

background process 8-12
bit fields A-2
bold characters 4-35
box function 4-34
buffered I/O
  creating 3-25
  described 3-24
  flushing a buffer 3-27
  returning a character 3-26
BUFSIZ, buffer size value 3-2
byte size A-5
bytes
  reading from a file 3-30
  reading from a pipe 7-4
  writing to a file 3-31
  writing to a pipe 7-5

C

C calling conventions C-1
C preprocessor 1-16
C source file, compilation 1-2
calling a program 6-2
calling sequence C-1
calloc function 9-2
calls
  header files 9-7
  include files 9-7
  message 9-19
    operation 9-21
    sample program 9-23
  semaphore 9-8
    operation 9-10
    sample program 9-15
    structures 9-9
  shared memory 9-25
case clause A-5
catching signals 8-4
CBREAK mode 4-39
cc command 1-1
cc options 1-26
character constants A-2
character functions 5-1
character pointer
  moving 3-35, 3-36
  moving to start 3-37
  reporting position 3-37
characters
  alphabetic 5-3
  alphanumeric 5-3
  ASCII 5-1
  control 5-3
  converting to ASCII 5-2
  converting to lowercase 5-6
  converting to uppercase 5-6
  decimal digit 5-4
  hexadecimal digit 5-4
  lowercase 5-6
  printable 5-5
  processing 5-1
  punctuation 5-5
  reading from a file 3-14
  reading from standard input 3-4
  uppercase 5-6
  whitespace 5-5
  writing to a file 3-17
  writing to standard output 3-7
chdir routine B-3

child process 6-6
chmod routine B-3
-cl option 1-10, 1-26
clear function 4-20
clearing screens 4-20, 4-29
clearing terminal modes 4-40
clearok function 4-37
close function 3-31, 7-5
closing a pipe 7-3
closing files 3-22, 3-31
clrtobot function 4-21
clrtoeol function 4-21
command line arguments 3-2
command line error messages D-1, D-3
common include files B-11
common routines B-1
common system-wide variables B-10
compilation 1-2
compilation error messages D-2, D-23
compiler error (code generation) D-3
compiler error messages
  command line D-1, D-3
  compilation D-2, D-23
  fatal D-1, D-18
  internal D-1, D-2
  warning D-1, D-7
compiler internal error messages D-1, D-2
compiler limits
  run-time A-7
compiler option summary 1-26
constants A-2
controlling the C preprocessor 1-16
creat routine B-3
creating faster programs 1-12
creating huge model programs 1-8
creating large model programs 1-7
creating middle model programs 1-7
creating object files, -c option 1-10
creating programs for DOS, -dos option 1-25
creating programs from object files 1-10
creating pure-text small model programs 1-7
creating small model programs 1-6
creating smaller programs 1-12
creating subwindows 4-23
creating windows 4-22
crmode function 4-39
ctype.h file 5-1
-Cu option 1-26
curses.h file 4-4

D

-D option 1-16, 1-27
data layout A-5
data types A-1
debugging
  preparing
    programs 1-14
  restrictions 3-2
decimal constants A-2
default output file 1-2
default terminal size 4-11
delay function 8-8
delaying signals 8-8

delch function 4-19
deleteln function 4-20
deleting characters 4-19, 4-28
deleting lines 4-20, 4-28
delwin function 4-34
disabling signals 8-2
DOS 1-25
-dos option 1-25, 1-26
duplicating a process 6-6

E

-E option 1-18, 1-27
echo function 4-40
ECHO mode 4-10
end-of-file testing 3-20
end-of-file value, EOF 3-1
endwin function 4-11
enum constants A-2
EOF, end-of-file value 3-1
-EP option 1-18, 1-27
erase function 4-20
errno variable 10-2
errno.h file 10-3
error messages 1-18
  command line D-1, D-3
  compilation D-2, D-23
  compiler D-1
  fatal D-1, D-18
  internal D-1, D-2
  warning D-1, D-7
errors
  catching signals 10-5
  delayed 10-4
  errno variable 10-2
  error constants 10-3
  error numbers 10-2
  printing error messages 10-3
  processing 10-1
  routine system I/O 10-4
  sharing resources 10-4
  signals 10-5
  standard error file 10-1
  system 10-4
  testing files 3-21
escape sequences A-3
exec routine B-4
execl function 6-3
execution profile 1-15
execv function 6-4
exit function 6-3

F

-F option 1-23, 1-27
far keyword 1-21
faster programs, creating 1-12
fatal error messages D-1, D-18
fclose function 3-22
feof function 3-20
ferror function 3-21
fflush function 3-27
fgetc function 3-14
fgets function 3-15
fields, bit A-2
file descriptors
  creating 3-29
  described 3-29
  freeing 3-31
  pipes 7-1
  predefined 3-29
file pointer type 3-2
file pointers
  creating 3-13
  defining 3-12
  described 3-12
  file descriptors 3-29

    FILE type 3-12
    freeing 3-22
    NULL value 3-12
    pipes 7-1
    predefined 3-13
    recreating 3-25
files
    buffers
        creating 3-25
        described 3-24
        emptying 3-27
        returning a character 3-26
    closing 3-22
    closing low-level access 3-31
    inherited by processes 6-8
    lockf 9-5
    opening 3-13
    opening for low-level access 3-29
    random access 3-35
    reading bytes 3-30
    reading characters 3-14
    reading formatted data 3-16
    reading records 3-15
    reading strings 3-15
    reopening 3-25
    testing end-of-file condition 3-20
    testing for errors 3-21
    writing bytes 3-31
    writing characters 3-17
    writing formatted output 3-19
    writing records 3-19
    writing strings 3-18
floating point constants A-2
flushing file buffers 3-27
fopen function 3-13
fopen routine B-5
fork function 6-6
formatted input
    reading from a file 3-16
    reading from a pipe 7-2
    reading from standard input 3-5
formatted output
    writing to a file 3-19
    writing to a pipe 7-2
    writing to standard output 3-8
fprintf function 3-19, 7-3
fputc function 3-17
fputs function 3-18
fread function 3-15
fread routine B-5
free function 9-4
freopen function 3-25
freopen routine B-5
-Fs option 1-12, 1-27
fscanf function 3-16, 7-2
fseek function 3-36
fseek routine B-5
fstat routine B-6
ftell function 3-37
ftell routine B-6
ftime routine B-6
function calls, counting 1-15
functions 3-12
fwrite function 3-19
fwrite routine B-7

G

getc function 3-14
getch function 4-15
getchar function 3-4
getpid routine B-7
gets function 3-5
getstr function 4-16
gettmode function 4-41
getyx function 4-36

H

header files 9-7
hexadecimal constants A-2
huge keyword 1-21
huge model programs 1-8

I

identifiers A-4
ignore default include directories, -X option 1-17
-il option 1-6, 1-27
inch function 4-32
include directories, defining 1-17
include file search, -I option 1-17
include files 2-1
include files, common B-11
initializing screen 4-7
initscr function 4-7
input 4-13
insch function 4-18
inserting characters 4-18, 4-27
inserting lines 4-19, 4-28
insertln function 4-19
integer sizes 1-29
integral constants A-2
internal error messages D-2
interrupt key 8-1
isalnum function 5-3
isalpha function 5-3
isascii function 5-1
iscntrl function 5-3
isdigit function 5-4
islower function 5-6
isprint function 5-5
ispunct function 5-5
isspace function 5-5
isupper function 5-6
isxdigit function 5-4
-Iu option 1-17, 1-27

K

-K option 1-14, 1-27
keywords 1-21, A-4

L

large model programs, creating 1-7
layout, data A-5
leaveok function 4-37
libcurses.a 4-4
libraries, using 1-9
library linking, -l option 1-11
libtermcap.a 4-4
limits, compiler

  - run-time A-7
- linking a program 1-11
- -ll option 1-11, 1-27
- lockf function 9-5, 9-6
- locking files 9-5
- locking routine B-7
- longjmp function 8-10
- longname function 4-44
- low-level functions
  - accessing files 3-28
  - described 3-28
  - file descriptors 3-29
  - random access 3-35
- lseek function 3-35, 9-5
- lseek routine B-7
- -Lu option 1-14, 1-27

M

- -M option 1-28
- macro definition, -D option 1-16
- macros, special I/O functions 3-2
- malloc function 9-2
- -Me option 1-21
- memory
  - allocating arrays 9-2
  - allocating dynamically 9-1
  - allocating variables 9-2
  - freeing allocated space 9-4
  - reallocating 9-3
- memory allocation functions 9-1
- memory model options 1-6
- memory models 1-29
- message calls 9-19
  - sample program 9-23
- -Mh option 1-8
- middle model programs, creating 1-7
- -Ml option 1-7
- -Mm option 1-7
- module names 1-30
- modules, segments, and groups 1-24
- mon.out file 1-15
- move function 4-17
- movement prefix, mv 4-38
- moving current position 4-17
- moving window position 4-27
- moving windows 4-32
- -Ms option 1-6
- multiple source files 1-3
- mvaddch function 4-38
- mvcur function 4-41
- mvwin function 4-32

N

- naming conventions 4-6
- naming output file 1-5
- -ND option 1-25, 1-28
- near keyword 1-21
- NEWLINE mode 4-40
- newwin function 4-22
- nl function 4-40
- -nl option 1-28
- -NM option 1-24, 1-28
- nocrmode function 4-40
- noecho function 4-40
- nonl function 4-40
- noraw function 4-40
- -NT option 1-25, 1-28
- null pointer value 3-2

O

O_APPEND 3-29
O_CREAT 3-29
O_RDONLY 3-29
O_RDWR 3-29
O_WRONLY 3-29
object files, creating 1-10
object files, using 1-9
octal constants A-2
-ol option 1-5, 1-28
open function 3-29, 9-5
open routine B-8
opening files 3-13, 3-29
opening pipes 7-2
operator, right-shift A-5
operator, sizeof A-5
optimization, -O option 1-12
option summary 1-26
-Ou option 1-12, 1-13
output 4-12
output file, naming, -o option 1-5
output optimization 1-12
output stripping 1-13
overlay function 4-31
overwrite function 4-31

P

parent process 6-6
pclose function 7-3
permissions 3-30
perror function 10-3
pipe function 7-3
pipes
  closing 7-3
  closing low-level access 7-5
  file descriptor 7-3
  file descriptors 7-1
  file pointer 7-2
  file pointers 7-1
  functions 7-1
  low-level between processes 7-8
  opening for low-level access 7-3
  opening to a new process 7-2
  reading bytes 7-4
  reading from 7-2
  shell pipe symbol 7-1
  writing bytes 7-5
  writing to 7-2
-pl option 1-15, 1-28
pointer operator A-5
pointer sizes 1-29
popen function 7-2
preprocessed source file 1-18
preprocessor 1-16
preprocessor control lines A-6
printf function 3-8
printing values 4-14, 4-24
printw function 4-14
process 6-1
process control functions 6-1
process ID
  ID 6-1
  termination status 6-3
processes
  background 8-12
  calling a system program 6-2
  child 6-6
  communication by pipe 7-2
  multiple copies 6-6
  overlaying 6-3

  parent 6-6
  restoring an execution state 8-10
  saving the execution state 8-10
  sklitting 6-6
  terminating 6-3
  termination status 6-8
  under shell control 6-6
  waiting 6-7
profiling code, -p option 1-15
programs from object files, creating 1-10
programs, invoking 3-2
-Pu option 1-18, 1-28
pure-text small model programs, creating 1-7
putc function 3-17
putchar function 3-7
puts function 3-7

Q

quit key 8-2

R

random access functions 3-35
raw function 4-40
RAW mode 4-10
read function 3-30, 7-4
read routine B-8
reading bytes 3-30
reading characters
  from keyboard 4-15
  from standard input file 4-25
  from window 4-32
  stream functions 3-14
reading formatted data 3-16
reading records 3-15
reading strings
  from keyboard 4-16
  from standard input file 4-26
  stream functions 3-15
reading values 4-16, 4-26
realloc function 9-3
records
  reading from a file 3-15
  writing to a file 3-19
redirection 1-18
redirection symbol
  input 3-9
  output 3-10
  pipe 3-10
refresh function 4-21
refreshing screens 4-21, 4-30
register variables A-4
removing stack probes, -K option 1-14
reopening files 3-25
reserved words A-4
restty function 4-42
return values, assembler C-2
rewind function 3-37
right-shift operator A-5
routines, common B-1
run-time limits A-7

S

savetty function 4-42
saving a preprocessed source file, -E and -P options 1-18
scanf function 3-5
scanw function 4-16
screen positions 4-3
screen processing
  /etc/termcap file 4-3
  adding characters 4-13, 4-24
  adding strings 4-14, 4-24
  bold characters 4-35
  clearing a screen 4-20, 4-29
  creating subwindows 4-23
  creating windows 4-22
  current position 4-3, 4-36
  curses.h file 4-4
  default terminal 4-9
  deleting a window 4-34
  deleting characters 4-19, 4-28
  deleting lines 4-20, 4-28
  described 4-1
  functions 4-1
  initializing 4-7
  inserting characters 4-18, 4-27
  inserting lines 4-19, 4-28
  libcurses.a file 4-4
  libtermcap.a file 4-4
  movement prefix 4-38
  moving a window 4-32
  moving the position 4-17, 4-27
  normal characters 4-36
  overlaying a window 4-31
  overwriting a window 4-31
  predefined names 4-2
  printing values 4-14, 4-24
  reading characters 4-15, 4-25
  reading strings 4-16, 4-26
  reading values 4-16, 4-26
  refreshing a screen 4-30
  refreshing the screen 4-21
  screen 4-3
  sgtty.h file 4-5
  standard screen 4-13
  terminal capabilities 4-3
  terminal cursor 4-41
  terminal modes 4-10, 4-39
  terminal size 4-11
  terminating 4-11
  using 4-7
  window 4-3
  window flags 4-10, 4-37
scrollok function 4-37
segment names 1-30
semaphore 9-8
  sample program 9-15
  structures 9-9
semaphores
setbuf function 3-25
setjmp function 8-10
setjmp.h file 8-1
setterm function 4-42

setting data segment name, -ND option 1-25
setting level of warnings, -W option 1-19
setting module name, -NM option 1-24
setting stack size, -F option 1-23
setting terminal modes 4-39
setting text segment name, -NT option 1-25
sgtty.h file 4-5
shared data
shared memory calls 9-25
shell, called as a separate process 6-6
shift operators A-5
signal function 8-2
signal routine B-8
signal.h file 8-1
signals
  catching 8-4, 10-5
  default action 8-3
  delaying an action 8-8
  disabling 8-2
  on program errors 10-5
  redefining 8-5
  restoring 8-3, 8-6
  SIG_DFL constant 8-2
  SIG_IGN constant 8-2
  SIGHANG constant 8-2
  SIGINT constant 8-2
  SIGQUIT constant 8-2
  to a child process 8-13
  to background processes 8-12
  with interactive programs 8-9
  with multiple processes 8-11
  with system functions 8-9
sizeof operator A-5
skrintf function 5-12
-sl option 1-13, 1-28
small model programs, creating 1-6
smaller programs, creating 1-12
source file 1-2
source file listing, -Fs option 1-12
sscanf function 5-12
stack order C-1
stack probes, removing 1-14
stack size, setting 1-23
standard error 3-4
standard error file pointer 3-13
standard error file, using 10-1
standard files
  described 3-4
  predefined file descriptors 3-29
  predefined file pointers 3-13
  reading and writing 3-4
  redirecting 3-4, 10-1
standard I/O file 3-1
standard I/O functions 3-1
standard input
  described 3-4
  reading 3-4
  reading characters 3-4
  reading formatted input 3-5
  reading strings 3-5
  redirecting 3-9, 3-10

standard input file pointer 3-13
standard output
  described 3-4
  redirecting 3-10
  writing 3-7
  writing characters 3-7
  writing formatted output 3-8
  writing strings 3-7
standard output file pointer 3-13
standend function 4-36
standout function 4-35
stat routine B-8
statements, switch A-5
stderr 3-1, 3-13
stdin 3-1, 3-13
stdio.h file 3-1, 3-13
stdout 3-1, 3-13
stopping a program 6-3
strcat function 5-7
strcmp function 5-8
strcpy function 5-9
stream functions
  accessing files 3-12
  accessing standard files 3-13
  file pointers 3-12
  random access 3-35
string functions 5-7
string literals A-2
strings
  comparing 5-8, 5-10
  concatenating 5-7, 5-10
  copying 5-9, 5-11
  length 5-9
  printing to 5-12
  processing 5-1
  reading from a file 3-15
  reading from standard input 3-5
  scanning 5-12
  writing to a file 3-18
  writing to standard output 3-7
strip symbol table, -s option 1-13
stripping files 3-11
strlen function 5-9
strncat function 5-10
strncmp function 5-10
strncpy function 5-11
structure member specification A-5
structures A-2
-Su option 1-14, 1-28
subwin function 4-23
switch statements A-5
symbol table, stripping 1-13
sys_errno array 10-3
system calls 9-7
system errors 10-4
system function 6-2
system include files
  a.out.h 2-6
  acct.h 2-6
  assert.h 2-6
  brk.h 2-6
  buf.h 2-6
  callo.h 2-6
  conf.h 2-6
  dir.h 2-6
  errno.h 2-7
  fblk.h 2-7
  file.h 2-7
  filsys.h 2-7
  ino.h 2-7
  inode.h 2-7
  iobuf.h 2-7
  ioctl.h 2-7
  ipc.h 2-7
  lock.h 2-7
  locking.h 2-7

machdep.h 2-7
map.h 2-7
mmu.h 2-7
mount.h 2-7
msg.h 2-8
param.h 2-8
proc.h 2-8
reg.h 2-8
relsym.h 2-8
relsym86.h 2-9
sd.h 2-9
sdu.h 2-9
sem.h 2-9
signal.h 2-9
sites.h 2-9
space.h 2-9
stat.h 2-9
sysinfo.h 2-9
sysmacros.h 2-10
systm.h 2-10
text.h 2-10
timeb.h 2-10
times.h 2-10
ttold.h 2-10
tty.h 2-10
types.h 2-10
ulimit.h 2-10
user.h 2-10
utsname.h 2-10
var.h 2-10
system programs, calling as a separate process 6-2
system resources 9-1
system routine B-9
system-wide variables, common B-10

T

TERM variable 4-9
termcap, capabilities from 4-9
terminal
  capabilities 4-3
  capability description 4-8
  clearing modes 4-40
  control 4-39
  cursor 4-41
  default size 4-11
  modes 4-10, 4-39
  reading names 4-44
  setting modes 4-39
  type 4-8
terminal screen 4-3
terminating screen processing 4-11
termination status
  described 6-8
  processes 6-3
testing end-of-file 3-20
testing file errors 3-21
toascii function 5-2
tolower function 5-6
touchwin function 4-33
toupper function 5-6
types, data A-1

U

umask routine B-9
unbuffered I/O
  creating 3-25
  described 3-24
  low-level functions 3-28
ungetc function 3-26

union member
 specification A-5
unions A-2
unistd.h file 9-5
unlink routine B-9
unlocking files 9-6
user include files
    ar.h 2-1
    assert.h 2-1
    core.h 2-1
    ctype.h 2-1
    curses.h 2-2
    dbm.h 2-2
    dial.h 2-2
    dumprestor.h 2-2
    errno.h 2-2
    execargs.h 2-2
    fcntl.h 2-2
    ftw.h 2-2
    grp.h 2-3
    macros.h 2-3
    malloc.h 2-3
    math.h 2-3
    memory.h 2-3
    mnttab.h 2-3
    mon.h 2-3
    pwd.h 2-4
    regexp.h 2-4
    sd.h 2-4
    search.h 2-4
    setjmp.h 2-4
    sgtty.h 2-4
    signal.h 2-4
    stand.h 2-4
    stdio.h 2-5
    string.h 2-5
    termio.h 2-5
    time.h 2-5
    unlstd.h 2-5
    ustat.h 2-5
    utmp.h 2-5
    values.h 2-5
    varagrgs.h 2-5
utime routine B-9

V

-V option 1-12, 1-28
variables
    allocating for
     arrays 9-2
    memory allocation 9-2
variables, common system
 wide B-10
version control, -V
 option 1-12

W

-W option D-1
waddch function 4-24
waddstr function 4-24
wait function 6-7
warning error
 messages D-1, D-7
warnings, setting
 level 1-19
wclear function 4-29
wclrtobot function 4-29
wclrtoeol function 4-30
wdelch function 4-28
wdeleteln function 4-28
werase function 4-29
wgetch function 4-25
wgetstr function 4-26
whitespace character 3-3
winch function 4-33
window
    border 4-34
    clearing 4-29
    creating 4-22
    deleting 4-34

  described 4-3
  flags 4-10
  moving 4-32
  overlaying 4-31
  overwriting 4-31
  position 4-3
  reading a character 4-32
  setting flags 4-37
  updating 4-33
winsch function 4-27
winsertln function 4-28
-wl option 1-28
wmove function 4-27
wprintw function 4-24
wrefresh function 4-30
write function 3-31, 7-5
write routine B-9
writing bytes 3-31
writing characters 3-17
writing formatted output 3-19
writing records 3-19
writing strings 3-18
wscanw function 4-26
wstandend function 4-36
wstandout function 4-35
-Wu option 1-19

X

-Xu option 1-17, 1-28

The Personal
Computer
Programming
Family

Reader's Comment Form

XENIX Operating System C Library Guide and Compiler Reference

SV21-8079

Your comments assist us in improving our publication; they are an important part of the input used for revisions.

IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please do not use this form for questions regarding setup, operation, or program support or for requests for additional publications. Instead, contact your authorized IBM Personal Computer dealer in your area.

Comments:

Tape

Do not staple

Tape

Fold here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM CORPORATION
DEPARTMENT 997
11400 BURNET ROAD
AUSTIN, TEXAS 78758

IBM

The Personal
Computer
Programming
Family

Reader's Comment Form

XENIX Operating System C Library Guide and Compiler Reference

SV21-8079

Your comments assist us in improving our publication; they are an important part of the input used for revisions.

IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please do not use this form for questions regarding setup, operation, or program support or for requests for additional publications. Instead, contact your authorized IBM Personal Computer dealer in your area.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM CORPORATION
DEPARTMENT 997
11400 BURNET ROAD
AUSTIN, TEXAS 78758

Fold here

Tape Do not staple Tape

© IBM Corporation 1986
All rights reserved.

International Business
Machines Corporation
P.O. Box 1328-W
Boca Raton,
Florida 33429-1328

Printed in the
United States of America

59X8657